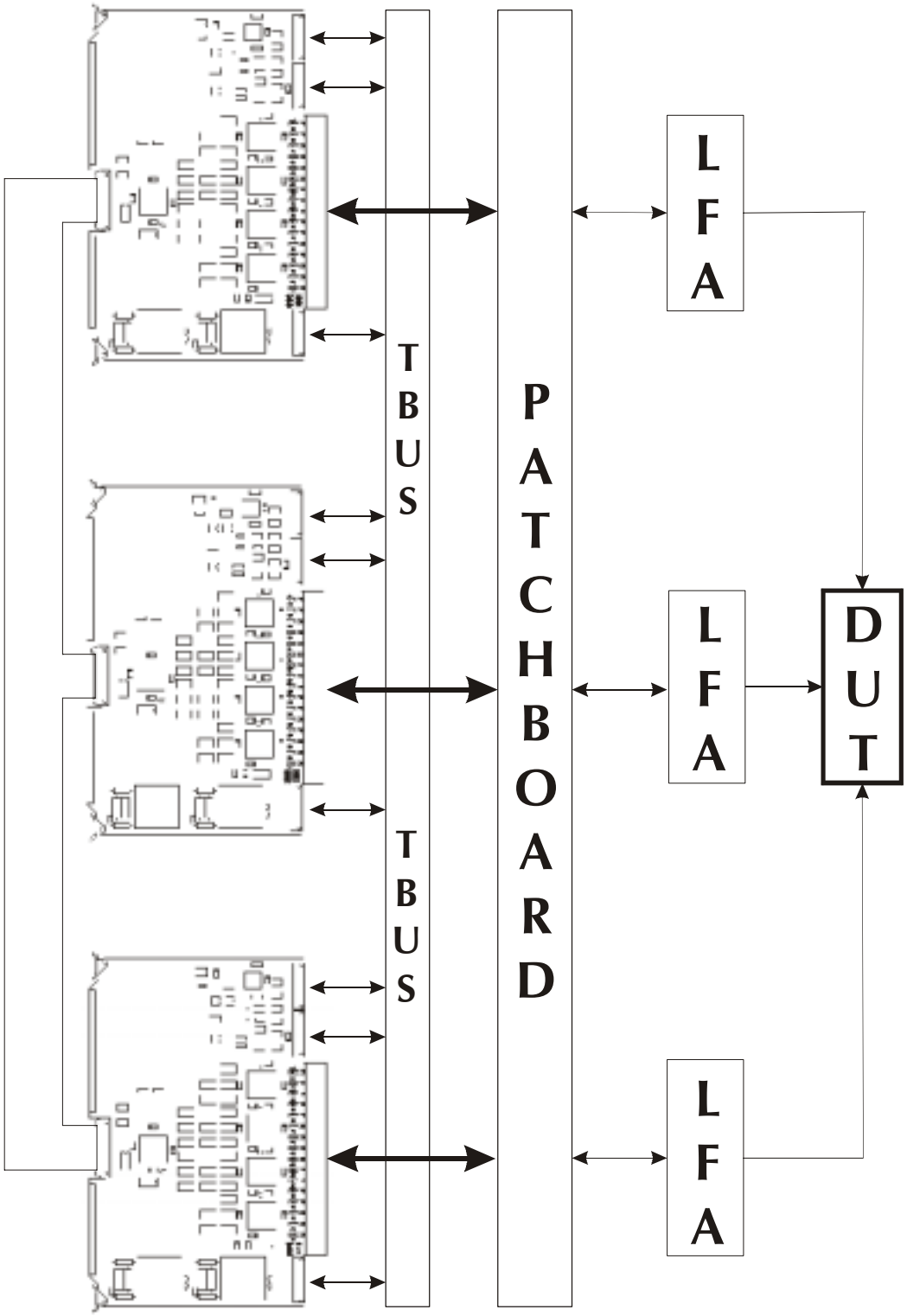
EPIO User Manual

PN#4200-0210

Version 2.0

Table of Contents

System Overview


TBUS
TBUS
PATCHBOARD
LFA
LFA
LFA
DUT

OVERVIEW

The Extended Pattern I/O System is the complete, high-speed digital test solution from Digalog Systems. Designed for the Digalog 2040D series of test equipment, it drives and receives high-speed digital data using on-board memory to buffer the incoming and outgoing test vectors. The EPIO system consists of the Testhead, EPIO Boards, and a series of Logic Family Adapters.

The first component of the system is the Testhead Bus (TBUS). This bus relays all information from the host computer to the cards in the Testhead. Every access, including configuration and data downloads, must use the TBUS. Access to the EPIO is provided through functional calls or the EPIO Pattern Editor (both covered later in this manual.)

The EPIO Board is the centerpiece of the digital test system. This Testhead add-in board buffers the data driven and received during a test. The board has a wide range of features, including adjustable cycle timing, evaluation of received data, and decision-making capabilities. The full feature list on the board is extensive, and a complete explanation follows in this manual.

The EPIO Ribbon Interconnect Cable is used to coordinate several EPIO boards into a single, cohesive unit. This allows the building of wider test systems with greater pin counts. Using the cable, several EPIO specific signals may be shared automatically, such as the common clock and test enable signals. Other shared signals include accumulator global carry lines and equality flags used for conditional instructions.

The EPIO Board to Device-Under-Test (DUT) connection is accomplished through the patchboard and a selected model of LFA. The primary function of the LFA is to adjust the digital logic levels of the EPIO System to those of the DUT. The target DUT and the complexity of its functional test dictate the model of LFA to use or design, with simpler models obviously being cheaper and faster.

Hardware

Hardware

The Extended Pattern I/O Board (EPIO) is designed to be an addition to the 2040 Series Testhead. The EPIO drives, receives, and evaluates high-speed digital signals using on-board memory to store the incoming and outgoing data.

The EPIO board is arranged into several sections. There are four I/O Formatters per EPIO board, each containing the Output Drivers, Receivers, Evaluation Units, and Timing Generators for eight digital channels. The Execution Controller, as its name implies, directs the flow of the execution from beginning to end. It manages the clocks and gates necessary to execute the patterns, and has the memory address counter, which indexes the current vector in onboard memory. The Execution Controller also houses four Trigger Matrix utility outputs with Timing Generators, which may be used to coordinate events with other Testhead resources. Each EPIO board also includes a DDS clock generator and the RAM necessary to hold the digital patterns.

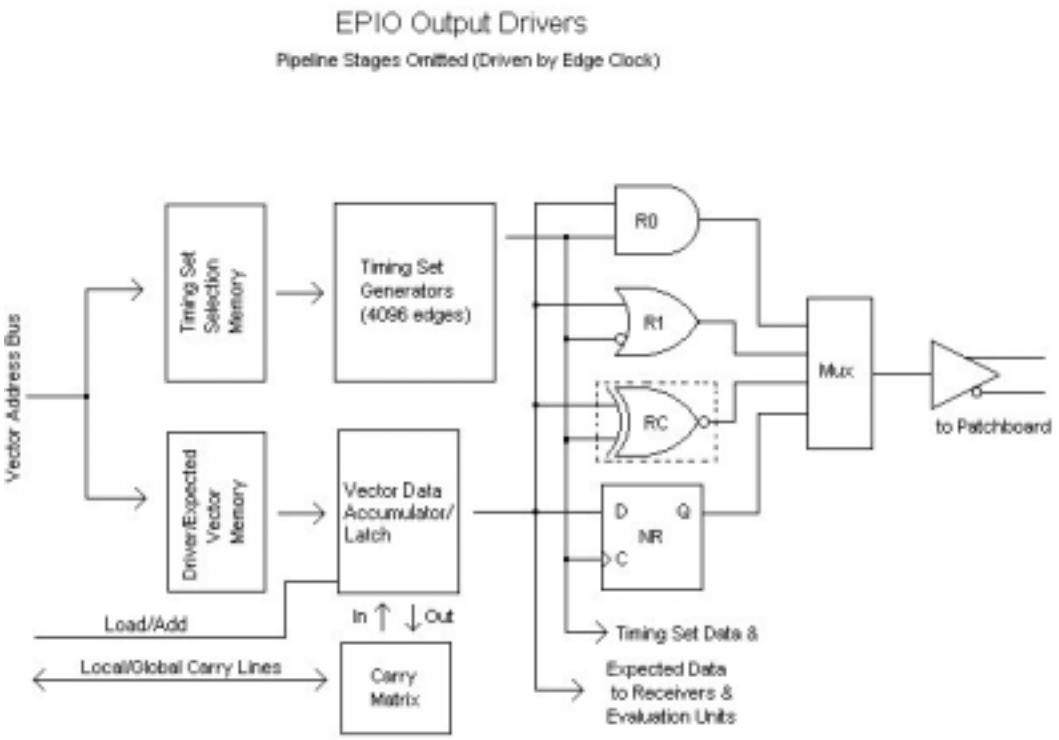
Channel Configuration

- 32 channels per board
- All channels may be configured as drivers or receivers (in groups of four)
- Twenty-eight channels per board may be configured as bi-directional (disable drivers on-the-fly, in groups of four)
- 512k memory depth

Each group of four channels on the EPIO board may be configured before execution as drivers or receivers. Up to twenty-eight of those channels (seven groups) may alternatively be configured as bi-directional, with the remaining four configured as full-time drivers for controlling the direction. Each of these four bits controls the three-state outputs of a pair of nibbles on the board. If none of the channels on a board are bi-directional, the remaining four channels may be used as ordinary drivers or receivers.

Drivers

- Each channel has user-programmable Driver Timing Sets
- NR, R0, R1, and RC pattern formats are available
- RZ pattern format supported using additional drivers as three-state control



Driver (or bi-directional) channels may be individually configured as having NR, R0, R1, or RC pattern formatted outputs. Timing set memory is used to provide the edge programming required. Bi-directional channels will behave as RZ-style formatted outputs with the timing sets of their control channels providing the edge timing.

Receivers and Evaluation Units

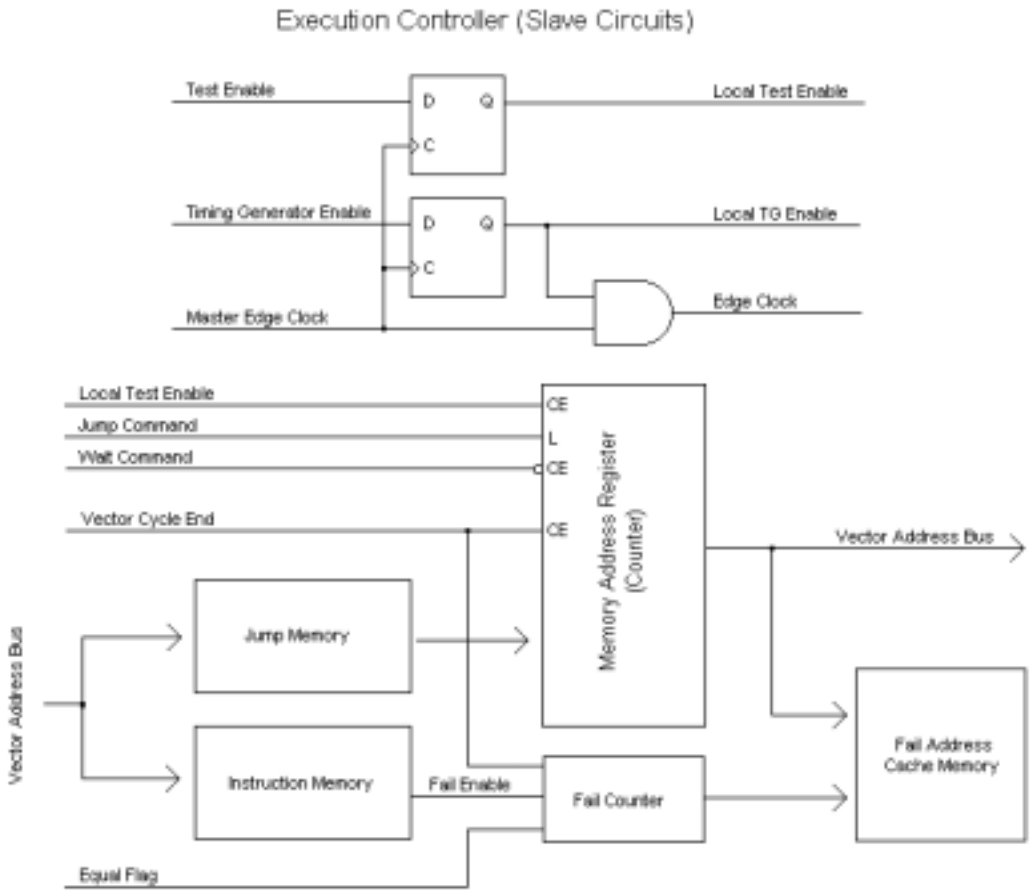
- Each channel has user-programmable Receiver Strobes
- Receivers capable of no-man's-land (NML) detection and evaluation
- Real-time evaluation units and failure flags on data received and NML detection

Every receiver and evaluation unit on the EPIO board operates full time. In practice, the Mask memory is used to effectively disable the channels not required for use with a particular vector pattern. The EPIO Pattern Editor software uses the Mask memory to disable the evaluation of any driver channels automatically. For receiver channels, the user can specify one of four possible comparison states for any channel/vector position during the test: High, Low, NML, or Don't Care.

The receiver timing pattern strobes the received data in on the last falling edge

of the pattern. This data is pipelined into memory and to the Evaluation Units, which mask off the undesired channels and compare incoming data to the expected data. The results provide an argument for several branching instructions of the Execution Controller. If the received data matches the expected, the Equal Flag will be a high value. Any non-matching data on any receiver will cause the Equal Flag to go low. In either state, the Equal Flag may be used as an argument to several instructions.

The receivers on the EPIO board are capable of detecting a high-impedance condition on the differential line connecting the EPIO channel to the LFA. This indicates that neither the LFA nor the EPIO is driving the differential lines. The initial intent of this ability is to build an LFA capable of detecting two voltage levels per product pin, with the intermediate zone being detected as a no-man's-land (NML) state. During a NML state, the LFA would not drive the differential lines back to the EPIO receivers, and the EPIO Board would recognize this condition properly. This state is also saved to memory and evaluated for the Equal Flag. Any complex threshold detection (such as adjustable or dual-level) must be provided by the design of the LFA.



Clocking

- Pattern vector rates up to 14MHz **(Preliminary)**
- Timing Set edge placement at 14.2ns (smallest possible edge-step size) (**Preliminary**)
- Single board sets master clock for EPIO subsystem

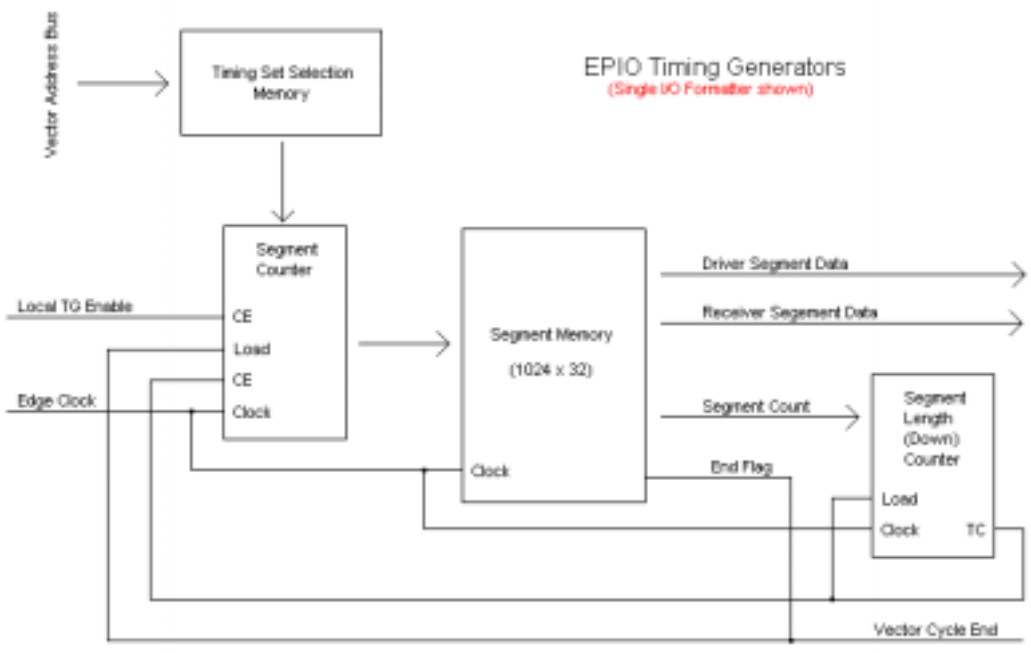
The master clock is programmed on a single EPIO board, somewhere between 300Hz and 70MHz **(Preliminary)**. This clock is shared among the boards in the system via a ribbon cable, providing a common synchronous reference. The edge clock used on individual EPIO boards is directly derived from the master clock.

The Master board will also provide the Timing Generator Enable and Test Counter Enable (see Execution Control). For all other EPIO boards in the system, the Timing Control Enable gate controls the generation and

distribution of the timing set edge clock on each individual board. The Test Counter Enable gate allows the vector patterns to advance or jump at the end of each timing set. The frequency of the master clock and edge clocks will be the same and will not change during pattern execution.

Vector data is advanced at the end of each timing set depending on the control instruction. Since timing sets can be variable lengths, the use of multiple timing sets allows vectors to be variable lengths.

Since the EPIO board will have a maximum programmable edge clock running at a maximum of 70MHz, it will not be possible to have an edge placement resolution finer than 14.2ns **(Preliminary Specification)**.



Vector Timing

- 256 timing sets available (maximum)
- 1024 total edges can be distributed among all timing sets defined for each byte
- Each channel can have its own timing patterns
- 14.2ns placement resolution of edges (**Preliminary Specification**)
- Keep alive signals (ability to hold vector output while waiting on a condition or reloading vector memories)

Timing sets are stored unique to each byte of all EPIO boards. Timing sets are programmed in "segments" that define the length of time between edge transitions for both drivers and receivers in the byte. At the end of each segment any, all, or none of the channels may change state (driver and receiver). Each byte has a total of 1024 segments that may be divided among several timing sets. Timing sets are created by stringing several segments together and finishing them with a programmed end segment flag. See Appendix A for a list of constants when defining timing sets.

For drivers, inactive (low) and active (high) states are programmed on each segment. An active region is the period of a timing pattern when the vector data is driven to the device-under-test. During inactive regions, the channel returns to the default state of its output format. An output format (NR, R0, R1, RC) for each driver is selected before patterns begin, and is not changeable on-the-fly. See Appendix B for examples of the driver output formats.

Receivers are implemented differently. The receivers will strobe whenever there is a high-to-low transition in their timing data. Since only the last data strobed by a timing set will be saved, it is redundant to have multiple receiver strobes within a single timing pattern. It is better to have a single receiver strobe to avoid confusion.

The edge clock counts the length of each segment, which may be up to 32768 counts long each. For example, if the edge clock is running at 10MHz, a segment of 1200 counts will complete in 120μs. If segments need to be longer than 32768 counts, simply place several consecutive segments together without changing states between them. Alternatively, the edge clock may be slowed, effectively reducing the number of counts required to span the same length of time, but also reducing the resolution available.

Each board's execution controller also has its own timing set area that must be programmed with segments, end segment flags, and Trigger Matrix edge definitions. The memory counter and vector pipelines are advanced when the end segment flag is reached in any timing set. Any low-to-high transition of the Trigger Matrix pattern (with asserted TM vector data) will cause a trigger to occur on a Trigger Matrix channel of the Testhead bus, provided that the appropriate setup commands have been previously executed.

Execution Control

- Instruction and Jump Memory runs in parallel to vector memory (512k)
- Two independent 24-Bit counters for looping instructions (nested or otherwise)
- Support for the Trigger Matrix Inputs and External Inputs
- Programmable Trigger Matrix outputs with Timing Set support

The Execution Controller guides the flow of every pattern. It is configured and armed before a pattern begins, and waits for the appropriate start signal. During execution, the test may contain instructions for:

- Conditional and unconditional jumping and looping on a group of pattern vectors
- Waiting on condition or a counter
- Algorithmic support

The pattern may be terminated with a variety of signals, any of which can be configured to leave the final vector on the output as a keep-alive signal to the device under test.

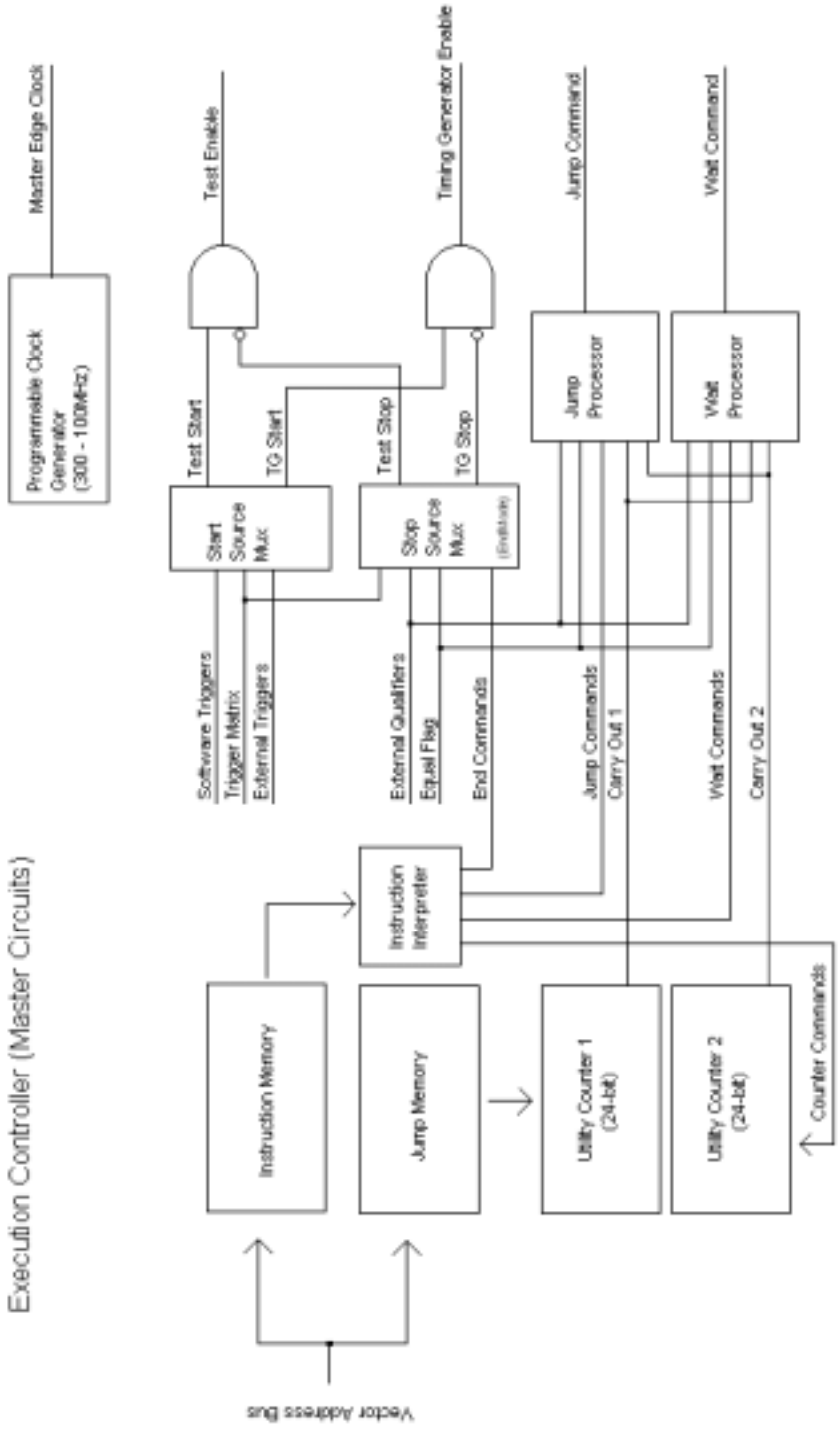
Program control and synchronization is distributed between boards. This includes:

- A master clock
- Timing Generator Enable
- Test Counter Enable
- A shared Equal Flag
- Algorithmic carry lines
- Jump and Wait commands

Pattern flow is started, controlled, and stopped by the master board.

Once the EPIO boards (including the master) are armed, start signals may come from the following sources:

- Software Command **EPStart** (the Timing Generators may be started in advance of the Test Counter by **EPEdgeStart**)
- Trigger Matrix
- External Input


Execution Controller (Master Circuits)
Vector Address Bus
Instruction Memory
Jump Memory
Instruction Interpreter
Utility Counter 1 (24-bit)
Utility Counter 2 (24-bit)
Counter Commands
Programmable Clock Generator (300 - 100MHz)
Master Edge Clock
Start Source Mux
Stop Source Mux
Test Enable
Timing Generator Enable
Test Start
TG Start
Test Stop
TG Stop
Jump Processor
Wait Processor
Jump Command
Wait Command
Software Triggers
Trigger Matrix
External Triggers
External Qualifiers
Equal Flag
End Commands
Jump Commands
Carry Out 1
Wait Commands
Carry Out 2

Stop signals may come from the following sources:

- Software Command **EPHalt** (for prematurely stopping the test)
- Vector Instruction (END, EOF, EOC, . . .)
- Trigger Matrix
- External Input

Programmed instructions can be given at any point in the pattern. Each vector has the possibility of one primary instruction and one or more secondary instructions being associated with it. However, some vectors will be off-limits due to the pipelining requirements of jump instructions. Please refer to Appendix C for information on pipeline depths that must be observed when programming jump, wait, and delay instructions.

The primary instructions are flow-oriented, containing commands for jumping, waiting and ending the test conditionally or unconditionally. For the conditional responses, several inputs are available for use. The equal flag is a global line, connected between all boards. This flag indicates whether the incoming response from the device under test is expected or not. The Evaluation Units are detailed in the section on receivers.

The external inputs are a group of signals that come from receiver channels on the master board. From these channels, a start trigger and six qualifiers may be selected. The start trigger may be configured to start a pattern burst, and the qualifiers may be used as conditional arguments for certain instructions. Two qualifiers each may be used to qualify End, Wait and Jump instructions for any test setup. The channel and polarity selections may be changed in between tests.

Trigger Matrix

- Start and Stop input triggers
- Start, Stop and Fail output triggers
- Up to four channels driven straight from control memory for synchronizing non-EPIO events to the tests running (with Timing Set Support)

Synchronization with Tester system events is an implemented feature on several Testhead cards. The mechanism for this feature is the Trigger Matrix. All Trigger Matrix-capable cards share access to the eight channels of the matrix, and allow triggered events on widely different subsystems. The EPIO supports

the Trigger Matrix in the following manner.
The Start signal, Stop signal, and Fail flag are all possible outputs from the master EPIO board's execution controller. Any or all of these signals could be routed to any channel on the Trigger Matrix. For instance, a waveform source board (properly equipped with the Trigger Matrix) could have its outputs started synchronously with the EPIO boards.

In addition, the EPIO board has four dedicated Trigger Matrix utility vectors, each with its own Timing Set. However, the active-low nature of the Trigger Matrix restricts the output format of any Trigger Matrix signal. High logic values on the vector, combined with inactive-to-active transitions of the timing set, will provide a trigger in the compliant format. (Specifically, this equates to an inverted R0 format.)

The EPIO execution controller circuitry may also use signals from the Trigger Matrix to Start or Stop the execution of a test.

Algorithmic Testing

- Four independently-controlled bytes per board
- Add instructions
- Global and Local routing lines for carry logic

Support for algorithmic testing is limited to addition instructions. Every bank of eight channels may be configured to add the vector in Driver/Expected memory to the vector already in the pipeline on an individual add instruction. These pipeline registers supply the output circuitry with driver data, and may supply the Evaluation Units with expected data. Subtraction operations are supported indirectly, by performing addition instructions with stored 2's compliment numbers.

Individual bytes on a board may be tied together into a continuous arithmetic unit using available local carry lines. Bytes and strings of bytes may be extended across multiple boards using four available global carry lines.
For example, a 32-bit address bus may be emulated as follows:

The least significant bit is channel 0 in byte 0 on board number 0 in the system. That byte's carry out is connected to the carry in on byte 1. Byte 1's carry out is connected to Global Bus 0, so it may connect off-board to byte 4 on board 1. Byte 4 is then wired locally to byte 5 to complete the 32-bit

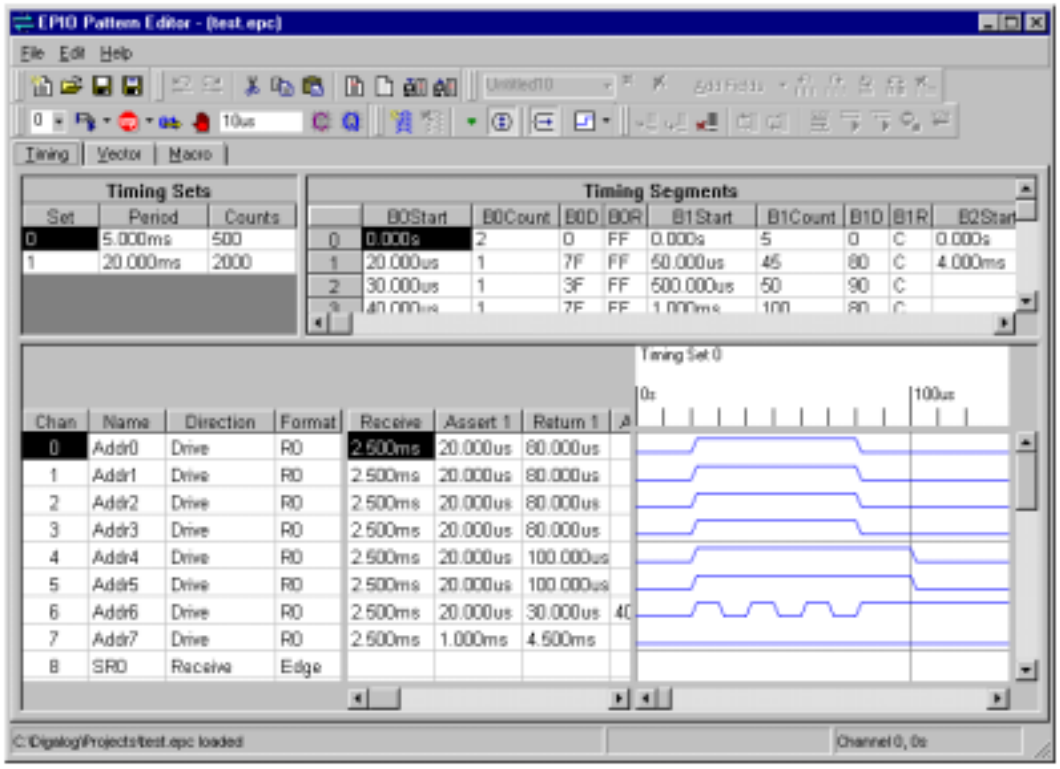
chain.

Byte 5 (MSB) ← Byte 4 ← (Global Bus 0) ← Byte 1 ← Byte 0 (LSB)

Propagation delays in the carry logic will prevent algorithmic operations from operating properly at the highest vector rates. The longer and more complex the carry chain is, the slower the circuit will be able to operate.

Pattern Editor

EPIO Pattern Editor



Pattern Editor

Description

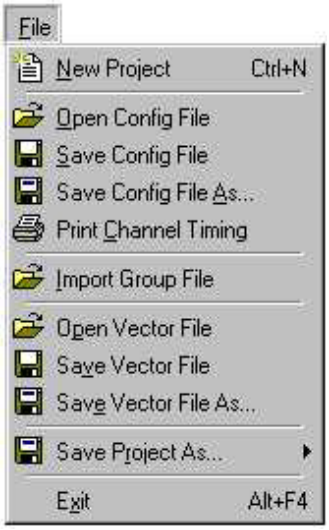
The **EPIO Pattern Editor** is an application for developing vector and algorithm-based digital patterns for use in functional testing. It consists of several tightly bound editors capable of developing, debugging, and deploying digital patterns within a functional test program. Each of the first three editors models a different portion of the EPIO hardware, and the last is a macro language editor to provide scripting support to the application.

- Configuration Editor
- Channel and Timing Editor
- Vector Editor
- Macro Editor

Menu Bar

File Menu

The File Menu contains options for creating and saving new projects, opening, saving, and renaming existing Configuration files, opening, saving and renaming Vector files, opening existing Vector groups, printing channel timing data, and exiting the Program Editor. Each will be briefly discussed here.



This selection resets the EPIO Pattern Editor to create a new project. Any Configuration or Vector files that have been loaded are flushed. If the configuration or vector files have changed, a prompt appears to save the changes.

This option displays a common dialog box for loading an existing Configuration (.epc) file. Opening a new Configuration file will cause the current Vector file to be flushed.

This option displays a common dialog box for saving the current Configuration file.

This option displays a common dialog box for renaming and saving the existing Configuration file.

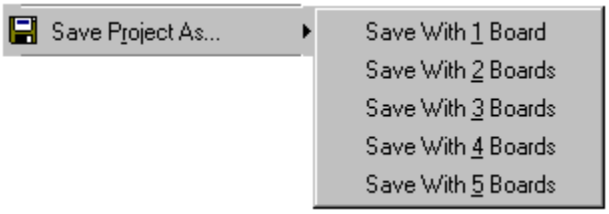
This option opens the Windows Print Dialog for printing a hard copy of the current timing sets. All channel setup information is printed including the channel direction, driver format, drive edges, and receiver strobe position.

This option allows the grouping setup of a different project to be imported into the current project.

This option displays a common dialog box for loading an existing Vector (.epv) file.

This option displays a common dialog box for saving the current Vector file.

This option displays a common dialog box for renaming and saving the current Vector file.



This menu option saves the current project for the specified number of boards. After saving, the project development will continue as normal with the addition (or subtraction) of the appropriate channel resources. This function is limited by the number of boards currently registered in the system.

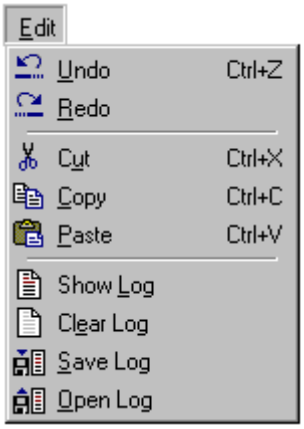
For technical reasons, the boards must be numbered sequentially from zero. For example, if a six-board system is configured for a three-board test, then boards 0, 1, and 2 must be used.

The next section of the menu is the “Most Recently Used” (MRU) section. It contains shortcuts to the last five Configuration and/or Vector files that were loaded.

This option closes the pattern editor. If any file has been modified, the operator will be prompted to save it before closure.

Edit Menu

The Edit Menu contains the standard Windows Cut, Copy, and Paste options. These options can also be used on code segments from within the Macro Editor and on individual steps from within the Vector Editor. This menu also includes Undo and Redo operations for manipulating edit operations. In addition, options for managing the Log Window are provided.



This selection undoes or negates the last operation performed in the Pattern Editor. The type of operation to be undone will be displayed adjacent to the Undo icon on the Edit Menu. For example, when the direction of a block of channels is changed, the Edit Menu would read “Undo - Change Direction.” This option will be “ghosted” until some edit is performed allowing an “undo” operation.

This selection redoes or reapplies the last operation undone in the Pattern Editor. The type of operation to be redone will be displayed adjacent to the Redo icon on the Edit Menu. For example, when the direction of a block of channels is changed, the Edit Menu would read “Undo - Change Direction.” If this option is undone, the Redo icon becomes active and “Redo - Change Direction” will be displayed. This option will be “ghosted” until some edit is performed allowing a “redo” operation

This option cuts or removes the selected data and places it on the internal Windows Clipboard. This option is useful in manipulating Visual Basic code in the Macro Editor. It can also be used on a single row or group of rows in the Vector Editor.

This option copies the selected data and places it on the internal Windows Clipboard. This option is useful in manipulating Visual Basic code in the Macro Editor. It can also be used on a single row or group of rows in the Vector Editor.

This option pastes data that has been Cut or Copied to the internal Windows Clipboard into the application. This option is quite useful in

manipulating Visual Basic code in the Macro Editor. It can also be used on a single row or group of rows in the Vector Editor.

This option is used to display or hide (toggle) the Log Window at the bottom of the Pattern Editor.

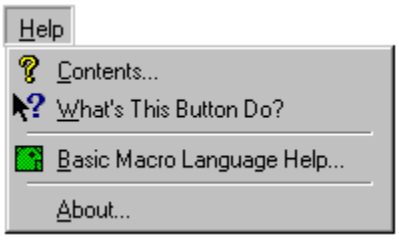
This option clears the contents of the log window. This action is irreversible.

This option saves the contents of the log window to a file. This information may be needed for project documentation, or may provide useful information to the Digalog Customer Support.

This option retrieves a previously saved log from a file.

Help Menu

The Pattern Editor Help File provides a "Contents" link to the Table Of Contents for the entire EPIO Help file. It also provides links to specific Help topics using the "What's This Button Do?" routine. In addition, a direct link to the Sax Basic Language Help file is included.



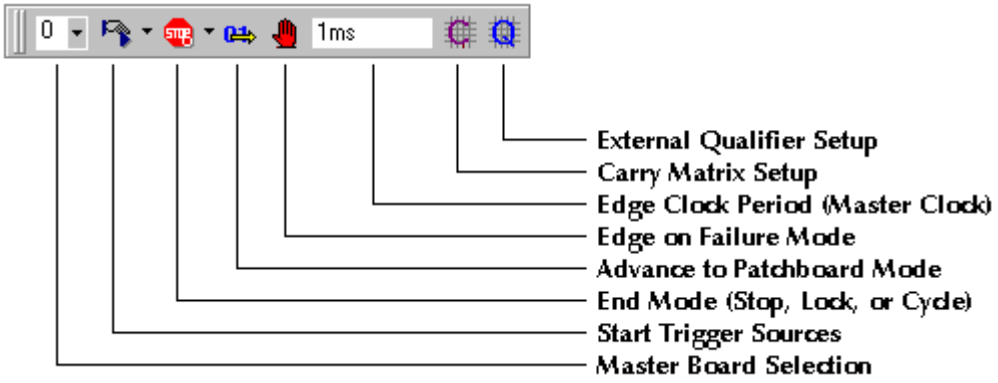
Organization

The program deals primarily with two file types. The first file type is the Configuration File, which contains the information entered in the Configuration and Channel/Timing Editors. Configuration Files have an ".epc" file extension by default. The second file type is the Vector File, which contains the data edited with the Vector Editor. Vector Files have an ".epv" file extension by default. Because channel and timing set data may be the same for several patterns of a single product, a single Configuration File might be used in conjunction with several Vector Files.

During the debug phase of the project, the Testhead resources must be accessed by an external utility or program. The power supplies, other Testhead cards, and the Logic Family Adapters can not be accessed through the Pattern Editor.

Once the files are created and debugged with the Pattern Editor, the patterns may be incorporated into an executive for production testing. When creating the executive, all EPIO Functional Calls (see next chapter) are available for use; however, two Functional Calls are especially important: **EPLoadVector** and **EPSetupConfig**. These functions encapsulate the abilities of many of the configuration and data-loading functions that only deal with an individual subsystem of any one EPIO board. The two files created by the EPIO Pattern Editor are loaded and parsed, and the appropriate individual functions are called. Once the boards have been configured and the data has been uploaded, the boards only need to be armed and the start signal given to run the test using the **EPArm** and **EPStart** functions. Once complete, the **EPStatus**, **EPFailAddresses**, and **EPResultData** functions may be used to discover the outcome of the tests. See Appendix E for an example of an executive routine to integrate and use these tests.

Configuration Editor



The Configuration Editor is actually located on the application's toolbar. Many of the EPIO's configurable options are accessible here, as well as access to the Carry Matrix setup screen.

- Master Board Selection
- Start Triggers
- End Mode Selection
- Edge placement resolution (Master Clock)

The Configuration Editor provides indirect access to the functional calls **EPEdgeClock**, **EPVectorGate**, and **EPExternalQualPolarity**, and specifies some of the arguments of **EPArm** and other functions (when debugging.) In addition,

the Carry Matrix setup window covers the available options when setting up an accumulator carry chain using the **EPCarryMatrix** function.

Master Board Selection

The EPIO boards operate in a master/slave type configuration, with the master board serving as the decision-making and timing coordinator for the system. Once the master is selected, the application will configureall other EPIO boards as slaves. The master is used as a parameter for several functional calls, and changes the way several calls behave. For instance, executing **EPArm** on a slave board has very different consequences than executing it on a master.

Start Trigger Sources

A start trigger, selected by **EPVectorGate**, allows the flow of pattern data to begin. Once the EPIO boards are armed, the start trigger may come from several sources. It may be provided by the user via the **EPStart** functional call. The debugger will automatically give this signal when the execute button is pressed. The start trigger may also come from receiver channels on the master board. The External Qualifier Setup dialog may be used to configure which two channels to use for external start inputs. A change in the external signal to the specified polarity will cause the test to begin.

End Mode Selection

Upon completion of any valid END instruction, the Timing Generators may be left in one of three states based on the *EndMode* parameter of the **EPArm** function. The Timing Generators may be Stopped, Locked on a final pattern (until the board is re-armed and started again), or Cycling. Cycle is only valid for conditional END instructions: EOE, EOU, EQ1, and EQ2. In the “Cycle on End” mode, the Timing Generator continues to run until the condition that caused the conditional END statement to be asserted is removed (a qualifier or an equal/unequal evaluation.)

Advance to Patchboard Mode

When arming the system before a burst, a choice may be made concerning the initial state of the drivers. After the drivers are cleared, they may optionally load the first vector’s data and present it to the patchboard. This allows the user to have known data on the patchboard when the test begins. Since this is usually the desired case, it is selected by default. This mode is ignored if the

system is currently in a “Lock-On-End” condition, where data is still active from a previous test.

Since this is a parameter of the **EPArm** functional call, it will need to be called appropriately when the test migrates to the executive environment (after the **EPSetupConfig** and **EPLoadVector** functions).

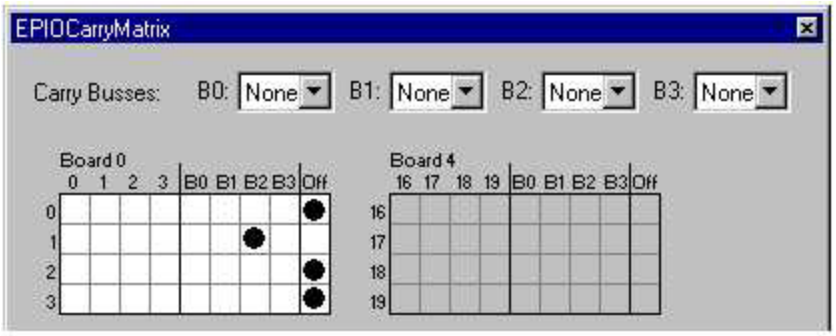
End On Failure Mode

The EPIO system may be put into an “End-on-Fail” mode when running a burst of vectors. During this burst, vectors that evaluate the received data as unequal to the expected and have the “Fail Enable” bit checked will trigger a spontaneous “End” instruction at that location. Yhis feature may prove useful when debugging tests.

Edge Placement Resolution

The edge placement resolution is the base synchronizing clock of the EPIO system and the Timing Generator. The Edge Clock is the responsibility of the master board, and it’s period determines the Timing Pattern edge placement resolution. The *Frequency* parameter of the **EPEdgeClock** function is derived from the reciprocal of the edge placement resolution. Note that the timing sets segment counts do not change when the edge placement resolution changes. If the resolution is made smaller, the length of time in which the timing set executes will be less as well.

Carry Matrix Setup



When configuring the Carry Matrix via the **EPCarryMatrixSetup** function, it is important to remember that each EPIO board has four byte-sized accumulators and local carry lines routed between them. Each accumulator handles eight board channels (of driver or receiver expected data), arranged in order (Byte 0 handling Channels 0 through 7, and so on). Each board also has

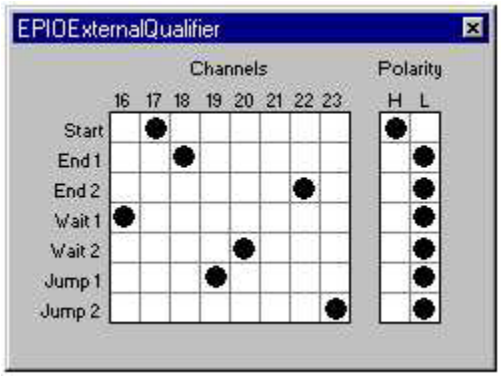
access to the Global Carry Busses, which may be routed to any destination as well.

At the top of the dialog, the sources of each of the four Global Carry Busses may be selected. Any byte-wide accumulator in the system may be used. Beneath, small grids are used to represent the Carry Matrix of each board. The left side of the grid lists the accumulator numbers (four per board) and the top represents the carry-in source (the other four bytes on the board, as well as the four Global Carry Busses.)

The picture on the previous page illustrates the configuration for a 32-bit accumulator, spanning two boards. From the most significant byte (Byte5) to the least (Byte0), the chain is:

Byte5 ← Byte4 ← (Global Bus 0) ← Byte1 ← Byte0

External Qualifier Setup



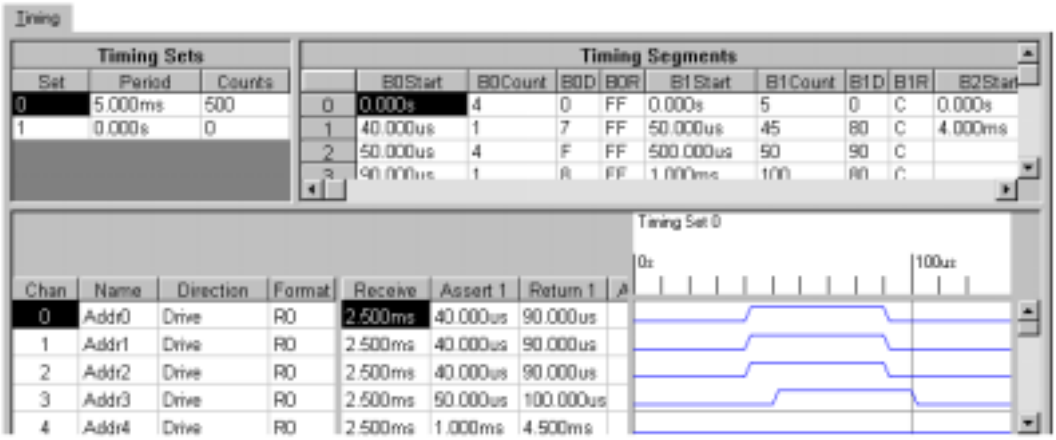
Several primary instructions rely on external inputs to provide conditional arguments. An external signal may also be the start trigger of the test. These qualifiers are linked to channels on the master board using this utility. Also select the desired "active" polarity of the signal. When programming the vectors, simply use the appropriate opcode (instruction) in the Vector Editor. The table below lists the opcodes that may be used with external inputs. The External Start Trigger may be selected using the Configuration Editor.

Start Trigger		Burst starts when trigger goes to selected state
Stop Qualifier 1	EQ1	Ends if qualifier is in selected state
Stop Qualifier 2	EQ2	Ends if qualifier is in selected state
Wait Qualifier 1	WQ1	Waits while qualifier is in selected state
Wait Qualifier 2	WQ2	Waits while qualifier is in selected state
Jump Qualifier 1	JQ1	Jumps if qualifier is in selected state
Jump Qualifier 2	JQ2	Jumps if qualifier is in selected state

The information entered into this dialog is used by the **EPExternalStartSetup**,

EPExternalEndSetup, **EPExternalWaitSetup**, and **EPExternalJumpSetup** functional calls.

Timing Editor



The Timing Editor configures the timing information used by individual EPIO channels as well as the direction, format, and labels of those channels.

Toolbar

The toolbar contains buttons for adding and removing whole timing sets, as well as configuring the timing pattern graphical editor.

Timing Set View

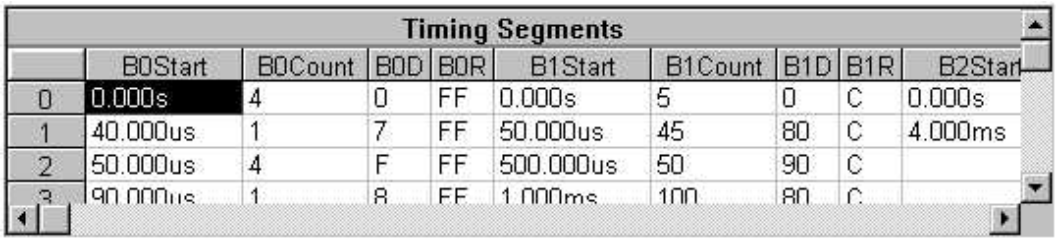
The small grid in the upper left corner of the editor allows the user to view the list of timing sets available for use in the Vector Editor. The lengths of the timing sets may also be edited here, using the overall period of the set or the total number of step-counts in the set. The step-counts multiplied by the edge placement resolution equals the overall period of the timing set.

Pressing this button will create a new timing set and add it to the end of the list. The new timing set will contain the minimum length necessary for operation, as defined by Appendix C. Additional segments will be added automatically when the test is executed from the debugger or the timing set is loaded from **EPSetupConfig** to meet

the minimum segment number requirement.

Pressing this button will delete the currently highlighted timing set. The user must be aware that vectors that made use of the deleted set will be pointing to a different or non-existant set after the operation. When deleting any timing set, caution should be exercised. The Undo operation will restore a deleted timing set.

Timing Segment View



Timing Segments

	B0Start	B0Count	B0D	B0R	B1Start	B1Count	B1D	B1R	B2Start
0	0.000s	4	0	FF	0.000s	5	0	C	0.000s
1	40.000us	1	7	FF	50.000us	45	80	C	4.000ms
2	50.000us	4	F	FF	500.000us	50	90	C	
3	90.000us	1	8	FF	1.000ms	100	80	C	

The timing segments programmed into the boards are viewable in the window to the immediate right. Each byte is displayed in every group of four columns. The starting time of the segment, length of the segment (in counts) and Driver and Receiver timing data patterns are displayed. This grid is made available as an informative view only; editing the timing sets must still be done in the graphical editor described on the below.

Note: If there are fewer than three segments, extra segments will be added to meet the “three segment requirements” listed in Appendix C before the timing sets are uploaded to the EPIO board. The extra segments are added such that the desired timing set is still intact.

As the complexity of the timing sets increases, the number of segments required by the overall timing sets also goes up. Please note that there are a limited number of segments available for use in the system. For more information on timing sets and the segments that they are constructed of, please refer to the Vector Timing area, under the Hardware section of this manual.

Channel and Timing Set Editor

The lower portion of the Timing Editor contains the channel area and graphical timing editor. This area enables several things to be edited.

- Channel Names
- Channel Directions
- Driver Output Format

Chan	Name	Direction	Format
0	Addr0	Drive	R0
1	Addr1	Drive	R0
2	Addr2	Drive	R0
3	Addr3	Drive	R0
4	Addr4	Drive	R0

This portion of the editor provides indirect access to **EPChannelSetup**, **EPDriverFormat**, **EPECTimingData**, and **EPIOTimingData**. Channel names are arbitrary and are for the user's reference within the Pattern Editor. All information is stored in the Configuration File, along with the configuration data, and may be used with multiple vector sets.

Each row of the editor represents one channel in the system and shows the relationship between the channels, their output formats, and their timing patterns. Every channel on every board registered in the system is listed. Besides an entry location for the channel name, users may select the direction and output format of the channel.

Channel Direction

Each group of four channels (nibble) may be separately configured for input or output for any test. The first seven nibbles on each board may also be configured for on-the-fly bi-directional operation. To control the vector-to-vector behavior of these bi-directional channels, the last four channels on the board will be automatically designated as outputs. The vector patterns for these four signals are used for three-stating or enabling the outputs of the other seven nibbles. The first three bits in the eighth nibble control the direction of the first six nibbles in pairs. The last bit in the eighth nibble controls the direction of the seventh nibble.

Control Channel	Channels Controlled (in groups of four)
28	{0..3}, {4..7}
29	{8..11}, {12..15}
30	{16..19}, {20..23}
31	{24..27}
60	{32..35}, {36..39}
Etc.	

The control nibble operates on the R0 output format, and it should be configured as such so the Logic Family Adapters (LFA) may also utilize these

signals for disabling output to the Device Under Test (DUT). For any vector, if the data asserted on the control channel is high, then the drivers will be active for the region outlined by the timing pattern of the control channel. If the data is low for the vector, then the drivers will not be active, and the channels may operate as receivers.

If any external start triggers or instruction qualifiers are to be used, then the 5th and/or 6th nibbles will need to be operating as receivers on the master board. This only needs to be done for nibbles containing qualifiers that are being used. The Timing Editor will **not** check to make sure this is done before execution.

Driver Output Format

The format of each output pin is configurable on the editor. This includes output pins and bi-directional pins during output vectors. The format determines how each Timing Pattern will be applied to the vectors. Supported formats are chosen from a short list of possibilities. See Appendix B for a table explaining the supported formats. The behavior of each is shown for low and high vector data. For receive-only channels, the format column is limited to edge mode only.

Edge Timing Editor

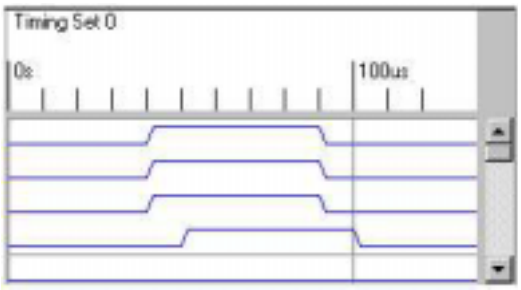
Receive	Assert 1	Return 1
2.500ms	40.000us	90.000us
2.500ms	40.000us	90.000us
2.500ms	40.000us	90.000us
2.500ms	50.000us	100.000us
2.500ms	1.000ms	4.500ms

The Edge Timing Editor allows the placement of channel receiver strobes and driver edges by text entry of the edge positions. These changes only affect the currently selected timing set. Receiver strobe times may be entered under the “Receive” column header. Upon entry, the strobe time for that channel (on the current timing set) is replaced by the new value. Any channel may have its receiver strobe specified. For drivers, the data may be masked off, if desired, in the vector editor. A strobe edge may be completely removed by pressing the <DEL> key, or by entering a time of “0s”.

Driver channel or bi-directional channels may have additional edges entered into the “Assert” and “Return” column pairs that follow. Each assert time marks the position that the drive vector data will be driven to the product. Each successive return time marks the position that the driver will return to its pin default. Pin defaults are specified by the driver format, such as Return to Zero (R0), Return to One (R1), and Return to Compliment (RC). For the No

Return format (NR), only the very first Assert edge will matter. Illustrations of the driver output formats in action may be found in Appendix B. The Edge Timing Editor may specify a limited number of Assert/Return pairs; additional edges must be defined in the Waveform Timing Editor.

Waveform Timing Editor



The last portion of the Timing Editor is a graphical editor that allows editing of the timing set by modifying the waveforms displayed with a mouse. Each channel is independently represented, and the editor displays one timing set at a time. Note that a change in the Edge Timing Editor will be reflected here, and vice-versa.

The driver and bi-directional channels have patterns that resemble a waveform, with low values representing “inactive” regions of the timing set, and high values representing “active” regions. These channels may have as many active and inactive regions as will fit into the length specified on the waveform editor.

Receiver strobe information is represented by a vertical colored bar on the pattern of the channel. Only one bar is allowed per channel, per timing set. Driver and bi-directional channels will have both types of indicators superimposed on the editor (waveform and strobe).

Pressing this button on the toolbar enables the Place Receiver Strobe editing mode. The user may click anywhere on any channel to position the receiver strobe for that channel within the timing pattern. Only one edge per channel per timing pattern can be placed.

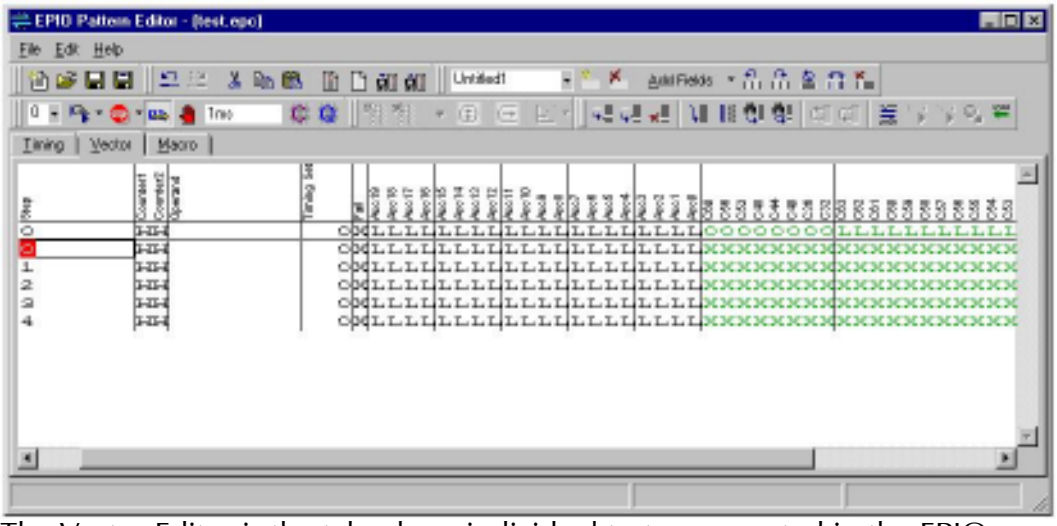
Pressing this button on the toolbar enables the Toggle Driver Edge editing mode. The user may click anywhere on a driver or bi-directional channel to toggle the state of a segment from active to inactive, or inactive to active. If there is no edge defined at the location, a new segment will be created, and the data for the remainder of the new segment will be toggled. Combined with the output format (See Appendix B), this pattern determines the output waveform.

Depressing this button may make editing with the Driver Toggle mode easier. With this option, toggling a segment will extend the new value to the end of the pattern. This makes it easy to fill active regions that may be hundreds of cells long.

The Undo and Redo options are also supported within the Channel and Timing Editor.

This control gives a drop-down list of choices to change the number of visible cells on the screen at a time. Using this ability, the user may see a large portion or all of wide Timing Set. Editing while at an extreme view will be difficult, and some placement precision will be lost.

Vector Editor



The Vector Editor is the tab where individual tests are created in the EPIO Pattern Editor. It contains all of the resources necessary to develop and debug vector patterns. The editor provides indirect access to the following functional calls:

- **EPPutDriverData**
- **EPPutExpectedData**
- **EPPutInstructionData**
- **EPPutExpectedNMLData**
- **EPPutTimingSelectData**
- **EPPutMaskData**
- **EPGetResultData**
- **EPGetResultNMLData**

There is support for direct updates of the memory on the EPIO boards during development allowing short download times when changes are made.

Toolbar

The toolbar for the Vector Editor contains buttons for adding and deleting vector steps as well as execution control (if running on a 2040 tester). The execution tool buttons are described in the Execution and Debugging section. There are five buttons for manipulating vector steps:

Insert a vector step before the current vector step.

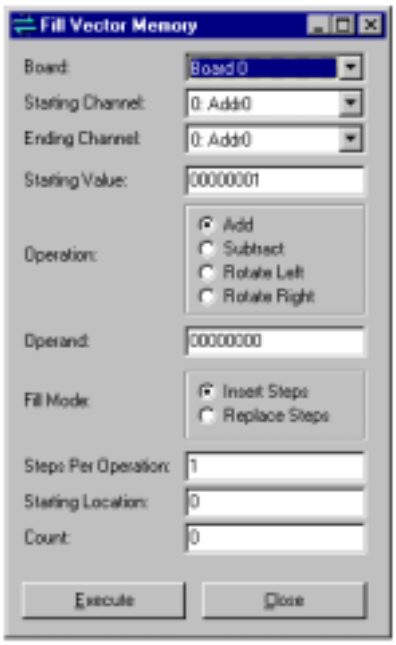
Insert a vector step after the current vector step.

Delete the current vector step.

The Goto Dialog allows the users to quickly jump to a vector step within the vector editor. The text box allows the choice to specify either the vector step number or a label (as specified in the label column). If the label entered is invalid, the a message box will be displayed. When a valid step or label is entered, the vector editor will jump to that step.

The fill dialog allows filling of EPIO vector data with patterns. It provides a large range of options to accomplish this. The fill dialog contains a fill accumulator that is used to perform algorithmic fills. There is a maximum size of 32-bits based on the EPIO channels per board. Here is a list of controls on the dialog and a description of their functions:

Board combo box - This combo box is used to select which EPIO board will be filled during the operation. It will contain every board contained in the project configuration.



Starting Channel combo box - This is the least significant bit used within the fill operation to be perfomed. Each channel is listed with its respective bit number and the channel name as defined in the timing editor.

Ending Channel combo box - This is the most significant bit used within the fill operation to be performed. Each channel is listed with its respective bit number and the channel name as defined in the timing editor.

Starting Value text box - The first vector step will be filled with this value. It is used for initializing the fill accumulator. It is specified in hexidecimal. The least significant bits are used based on the difference between the starting channel and ending channel. For example, if channel 0 is the starting channel and channel 3 is the ending channel, then only the lowest 4 bits will be used in the operation. The other bits are ignored.

Operation option buttons - The operation defines what will be done with the fill accumulator. When an operation is performed, it uses that value specified in the Operand text box. There are four operations that can be performed on the fill accumulator:

> **Add** - Add the value specified in the Operand field to the value in the fill accumulator for each operation. It will only use the lowest bits of the Operand field based on the difference of the Ending Channel and Starting Channel fields.
>
> **Subtract** - Subtract the value specified in the Operand field to the value in the fill accumulator for each operation. It will only use the lowest bits of the Operand field based on the difference of the Ending Channel and Starting Channel fields.

Rotate Left - This will rotate the fill accumulator to the left a number of bits as specified in the Operand field.

Rotate Right - This will "rotate right" the fill accumulator a number of bits as specified in the Operand field.

Operand text box - The Operand text box is where the value used for each operation is specified. The value is in hexidecimal, and only the number of bits based on the Starting Channel and Ending Channel are used.

Fill Mode option buttons - There are two Fill Mode options:

Insert Steps - Inserts the specified number of steps.

Replace Steps - Replaces the specified number of steps. Replace steps will only fill up to the maximum number of steps currently in the vector file loaded.

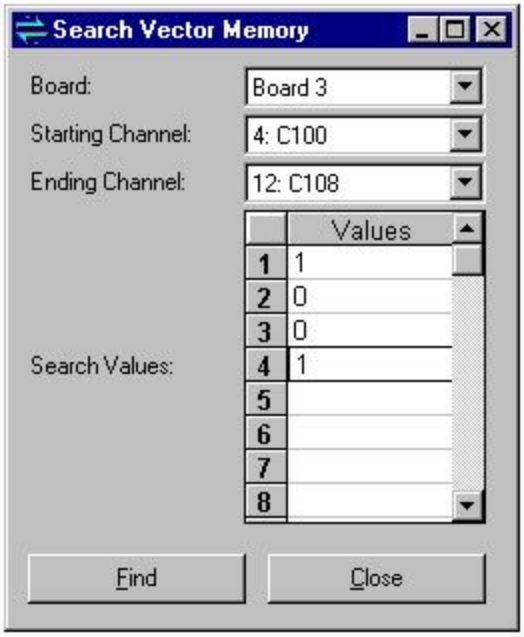
Steps Per Operation text box - This entry allows a number of steps to be filled before an operation is performed. For example, if 5 is specified in the field with an add operation specified in the operation field, every 5 steps will contain the same value, and then the operation will be performed. Then the process will be repeated.

Starting Location text box - This is the vector step that is initially used.

Count text box - This is the number of times to perform the operation. For example, if steps per operation is set to 5 and the count is specified as 100, the starting location is set to 0 and 500 vector steps will be filled from step 0 to 499.

Search Vector Memory - This option displays a dialog for entering search information for the Vector Editor, as shown on the next page. The "Board" listbox contains all of the available EPIO boards in the system. The "Starting & Ending Channel" listboxes contain all of the available channels on the board selected in the "Board" listbox. The "Search Values" grid is used to enter the pattern being sought by the programmer. If or when this bit pattern is found, the values will be

highlighted in the Vector Editor. As an example, the dialog above indicates that the programmer wishes to search channels 100 to 108 (on Board #3) for the bit pattern "1001".



If the search needs to be repeated, use the "Repeat Search" option. If a new set of values or a different location needs to be searched, use this option again and enter the new search information.

Repeat Search - This option repeats a memory search using the search parameters entered in the "Search Vector Memory" routine.

Columns

The Vector Editor is divided into columns containing various parts of the **EPIO** board. Here is a list of columns:

Step

This is the step number in the vector pattern. It is primarily used during debug when moving to each of the failed addresses.

Label

This is a label used for JUMP instructions and the **EPGetLabelStep** function. Each vector step that needs to be "jumped" to requires a label in order to map the jump locations in the EPIO memory. Labels are up to 8 characters in length and should start with a letter and can be followed by letters and numbers. Labels are case sensitive. After the **Label** field is referenced to a vector step, this information is used in a **EPPutInstructionData** functional call. Labels are not required if the vector is not to be used as a target location for a JUMP instruction.

Comment

This is a 16 character field to comment the vector pattern step. It is useful for

documenting vector steps or pattern sections and is not used anywhere else. Any valid character that can be typed can be used.

Opcode
This is the EPIO execution controller opcode used to control the flow of exectuion. The codes are typed in directly and are case-sensitive. Here is a list of valid opcodes and their corresponding primary instructions:

- **NOP** - No Operation
- **END** - End (Unconditional)
- **EOU** - End On Unequal
- **EOE** - End On Equal
- **EQ1** - End On Qualifier 1
- **EQ2** - End On Qualifier 2
- **EOF** - End On Fail Flag (Set by any previous vector failures)
- **EIM** - End Immediate (no receiver pipeline flush, no buffer vectors necessary)
- **WOU** - Wait On Unequal
- **WOE** - Wait On Equal
- **WQ1** - Wait On Qualifier 1
- **WQ2** - Wait On Qualifier 2
- **DC1** - Delay On Counter 1 (Requires a Counter 1 “Up” or “Down” Instruction.)
- **DC2** - Delay On Counter 2 (Requires a Counter 2 “Up” or “Down” Instruction.)
- **JMP** - Jump (operand contains the **Label** to jump to)
- **JOU** - Jump On Unequal (operand contains the **Label** to jump to)
- **JOE** - Jump On Equal (operand contains the **Label** to jump to)
- **JC1** - Jump On carry clear of Counter 1 (operand contains a **Label** to jump to; equires a Counter 1 “Up” or “Down” Instruction.)
- **JC2** - Jump On carry clear of Counter 2 (operand contains a **Label** to jump to; requires a Counter 2 “Up” or “Down” Instruction.)
- **JQ1** - Jump On Qualifier 1 (operand contains a **Label** to jump to)
- **JQ2** - Jump On Qualifier 2 (operand contains a **Label** to jump to)

The **Opcode** is used by the **EPPutInstructionData** functional call.

Counter1
This is the Utility Counter 1 control field. There are 4 modes:

- 'Load' - Load the utility counter with the contents of the **Operand** field. The operand field must be a hexadecimal number. The '**L**' key pressed in this field sets the 'Load' mode.
- 'Hold' - Hold the counter at its current value. The '**H**' key pressed in this field sets the 'Hold' mode.
- 'Up' - Increment the counter by one. The '**U**' key pressed in this field sets the 'Up' mode.
- 'Down' - Decrement the counter by one. The '**D**' key pressed in this field sets the 'Down' mode.

This is used by the **EPPutInstructionData** functional call.

Counter2
This is the Utility Counter 2 control field. The modes and key input is the same as the Counter 1 field. This is used by the **EPPutInstructionData** functional call.

Operand
This is used for the JUMP opcodes or the load values for the utility counters. If a JUMP instruction is used, this field must be a non-empty valid label defined in the Label field. If a 'Load' in either or both utility counters is used then this field is formatted as a hexadecimal number. This field is used by the **EPPutInstructionData** functional call.
Timing Set
This is the timing set to be used for the vector. To prevent errors, the timing set ***must*** exist in the Timing Editor. This field is used by the **EPPutTimingSelectData** functional call.

Fail
Enable fail checking for this step. For the current step, a '**F**' typed into this field enables fail checking whereas a '**X**' typed into this field disables fail checking. This field is used by the **EPPutInstructionData** functional call.

Acc0 to AccX
The **AccX** fields control the accumulators of the EPIO board. Each board contains 4 **AccX** fields. For the current step, a '**L**' typed into this field loads the

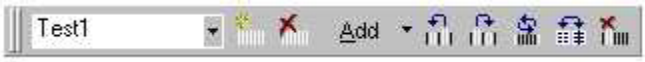
accumulator with the contents of the EPIO driver/expected memory. This is the default value and keeps vectors flowing directly from memory. An '**A**' typed into this field adds the contents of the EPIO driver/expected memory to the value contained in the accumulators. Accumulator operations are performed on a byte basis and can be combined with other accumulators by using the Carry Matrix Setup dialog from the Configuration Editor. This field is used by the **EPPutInstructionData** functional call.

Vector Data

The **ChannelX** fields are used for the EPIO driver/receiver memory. They define what is being driven and what is expected. This is based on each channel's configuration defined in the Channel and Timing Editor as well as the direction control bits if the channel is defined as being bi-directional. The data located in these columns are processed and used in the functional calls: **EPPutDriverData, EPPutExpectedData, EPPutMaskData,** and **EPPutExpectedNMLData**. The following keys are used in these columns:

- '**1**' or '**H**' - will drive or expect a high logic level (based on configuration settings). Both will work on any channel configuration (input, output, bi-directional).
- '**0**' or '**L**' - will drive or expect a low logic level (based on configuration settings). Both will work on any channel configuration (input, output, bi-directional).
- '**N**' - expecting to receive *no-mans-land* logic level. This key is ignored when a channel is defined as an output.
- **X**' - will set the output to a low logic level when the channel is an output. When the channel is an input the data is masked for comparison opcodes. For bi-directional channels, the direction bit defines a high-impedance state, not the channel itself.

Grouping Toolbar



The Grouping Toolbar allows you to create different column arrangements of your vector data, for viewing comfort and convenience. Several tools on this menu allow you to add, move, and delete the full range of columns. The groupings created are automatically saved in a group file (.epg) when the configuration file (.epc) is saved and when exiting the editor anytime thereafter.

The first tool is the **Select Group** combo box. When the entry is edited like a text box, it will change the name of the current grouping scheme. When the combo box is dropped down, it will select a new grouping scheme. The first group is "Default". When this is selected, the column editing tools are disabled. The "Default" group creates the execution fields, the accumulator control fields, and all bits of all boards moving from MSB to LSB (EPIO channel) of the highest board going to the lowest board. If you want use the "Default" group as a template, you can rename the "Default" group to something else. This will enable the column editing tools. You can also rename new groups created. They are usually labeled "UntitledXX" with XX being a number. A group name must be unique.

The **Create New Vector Group** tool will create a new empty group with a default name. When a new group is created, the only field displayed will be the step number. Fields can be added with the Add Pulldown Menu. The name of the group can be directly edited with the Select Group Combo Box.

The **Delete Vector Group** tool will delete an existing group. You cannot delete the "Default" group.

The **Add Fields** pulldown menu allows appending columns to the right of the existing ones. The columns can be moved around (as described below). The menu contains four entries:

Add Execution Fields - This adds the all the execution fields: Label, Comment, Opcode, Counter1, Counter2, Operand, Timing Set, and Fail.

Add Accumulator Fields - This adds the accumulator control fields for all bytes of all boards starting from highest byte, highest board to lowest byte, lowest board.

Add Bit Fields - This is a pullout menu that has each board that is configured. Each menu item will add the selected board's channels from MSB to LSB, formatted as individual bits.

Add Nibble Fields - This is a pullout menu that has each board that is configured. This menu item will add the selected board's

channels from MSB to LSB, grouped together in nibbles.

When an area is selected in the Vector Editor, the **Shift Selected Fields Left** tool will shift all of the columns within the selection to the left.

When an area is selected in the Vector Editor, the **Shift Selected Fields Right** tool will shift all of the columns within the selection to the right.

When multiple columns are selected in the Vector Editor, selecting the **Reverse Selected Fields** button will swap all of their positions. This function is useful for reversing the order of channels; however, it may be used on any selected column.

When an area is selected in the vector editor and the **Toggle Selected Fields** is pressed, all of the columns within the selection will have the separator toggled from visible to invisible, depending on what it is when the tool is clicked.

When an area is selected in the vector editor, the **Delete Selected Fields** button will delete all of the columns within the selection.

Execution and Debugging

The execution tools allow you to execute the current test without saving configuration and vector data to files and leaving the application. The tools outlined below are not available to development systems that are not connected to a Testhead with EPIO boards installed.

Synchronizes the vector memory on every EPIO board with the data in the Vector Editor. The driver, expected, mask, instruction, and jump memories are updated continuously while this button is depressed. This button must be depressed to enable the following execution options:

Updates the hardware with the current configuration and timing data, arms the boards, and executes the test from the beginning. When complete, results are retrieved from the boards and displayed in the top row of the Vector Editor and in the Log Window.

Executes the current test, as before, but begins from the step highlighted in the Vector Editor.

Stops the currently running execution as soon as possible.

Jumps to the previous vector error from the current position. It is not enabled until a test has executed and results have been retrieved.

Jumps to the next vector error from the current position. It is not enabled until a test has executed and results have been retrieved.

Selecting this button synchronizes the Vector Editor with the vector memory on every EPIO board. The driver, expected, mask, result and timing set select memories are retrieved from the hardware and displayed. This function is meant to be used if the data is being changed using an external program, or to update the results displayed if an external executive is executing bursts.

This function will retrieve data, starting at a board memory offset of 0 for the length of the currently loaded vector file. This implies that the test is setup similarly between the Pattern Editor and whatever external executive is altering the test data.

Other Functions

Edit Menu and Toolbar

The edit menu and toolbar can also operate within the Vector Editor. The tools operate as follows:

Undo the last Vector Editor or Timing Editor operation. There are separate undo/redo lists for each editor. When either tab is visible, the respective undo/redo list is used.

Redo the last undone Vector Editor or Timing Editor operation. This works the same as described for the undo operation.

When a selection is made in the Vector Editor, the number of steps selected are removed and placed in the cut buffer.

When a selection is made in the Vector Editor, the number of steps selected are copied into the cut buffer.

Insert the steps from the cut buffer before the top of the current selection.

Log Window

The log window is an output window that shows certain events as they happen in the system. This may be useful for keeping the user informed during a process, such as pointing out errors in file as they load. Limited access to the log window is also provided through the Macro Editor. Otherwise, many of the Pattern Editor's built-in commands (such as file loading and saving) automatically use this window for standard output.

Toggles the log window visible or invisible.

Clears the contents of the log window. This action is irreversible.

Saves the contents of the log window to a file. This information may be needed for project documentation, or may provide useful information to the Digalog Customer Support.

Retrieves a previously saved log from a file.

Macro Editor

The Macro Editor is used to process information within the **EPIO Pattern Editor**. It allows manipulation of the Project including configuration, timing, and vector data. The primary purpose is to create vector pattern importers for various simulation programs. It contains the following functions:

- **ClearLog()** - clears the log window.
- **AddLog(LogMsg As String, Optional Color As Long = vbBlack)** - adds a line to the log window making it visible if its hidden.
- **NewProject()** - starts a new project.
- **SaveConfigFile(FileName As String)** - saves the configuration file based on the file name passed either in the current directory or the

path specified by the **FileName** parameter.

- **SaveVectorFile(FileName As String)** - saves the vector file based on the filename passed either in the current directory or the path specified by the **FileName** parameter.

Note: Any operations done in the Macro Editor are not checked to see if the data has been modified. If you exit the editor, create a new project, or load an existsing project without saving, the information will be lost. The **EPIO Pattern Editor** will **not** post a warning message about unsaved data when macro operations are executed.

There is one object accessible to the Macro Editor: **Project**. This contains the configuration, timing, and vector data of the current project. It contains many properties. Consult the macros object browser for a list.

There is a menu available in the Macro Editor window that is used for loading and saving macro files. A click with the right mouse button in the Macro Editor window will bring up the menu. All menu operations accessible to the editor are available through this menu. The Undo and Redo tool buttons in the main **EPIO Pattern Editor** window will also work with the Macro Editor when editing code.

Functional Calls

EPArm

This functional call is the final EPIO function necessary to prepare for the start of execution. It is used after configuring the EPIO board's clocks, gates, and mode after all of the required vector data has been uploaded. It must be called once for every board in the system since it enables the appropriate signals and loads the memory address register for each board. Since the *MasterEnable* parameter enables one board to drive several control signals onto a bus, the board that will be the master should be armed last to avoid bus contention.

The master board will be generating Timing Generator and Test Counter Enable gates between selected Start and Stop signals. The circuit is enabled for output and armed for the Start trigger specified by **EPVectorGate**. Jump and Wait commands are also generated here, and the Fail flag is reset. This is usually the appropriate board to use as the argument for the **EPStatus** functional call.

Before executing tests, the data that occupies the driver and instruction pipelines may be unknown. To eliminate problems, the driver and instruction pipelines are cleared before the board is armed. The data at the patchboard will be the format default (low for R0, high for R1, low for RC and NR.) Any bi-directional channels used will also be set to receive. Alternatively, the first vector of a test may be pre-loaded and clocked until it appears on the outputs of the board. Once the test starts, the data is updated as usual. Either of these methods prevents the product from seeing unknown data.

An *EndMode* parameter will specify the state of the Timing Generators after a valid END command. The Timing Generators may be stopped thereby discontinuing any change in the outputs from the board. It may lock on the final vector, and the Edge Clock would keep applying the same Timing Set to the final vector. This may be useful in providing "keep-alive" signals to a DUT between tests. In this mode, the board must be re-armed with **EPArmDuringLock** to run further tests. **EPHalt** may also be used to stop the Timing Generators. Finally, it may cycle on the final vector when a conditional END is reached. The boards will stop normally when the condition is removed, as described above.

Visual Basic Declaration: **epio32.bas**

Public Sub EPArm(ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal MasterEnable As Integer, ByVal DriverClear As Integer, ByVal EndMode As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPArm (int16 boardNumber, int32 startAddress, int16 masterEnable, int16 driverClear, int16 endMode);

Call EPArm(BoardNumber, StartAddress, MasterEnable, DriverClear, EndMode)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
- = 0 EPIO board 0.
- = 1 EPIO board 1.
  Etc. to board 7.

StartAddress Any valid memory location to set the address counter to. It should be the location of the first vector in the test to be started.

MasterEnable Signifies if this board is controlling the execution flow.
- = 0 This board will be a slave board.
- = 1 This board will provide the command processing.

DriverClear This option will clear any data that occupies the driver pipelines from power-up or from previous tests. This prevents the product from seeing unknown data until the first vector reaches the end of the pipeline. It is recommended that option #1 be used for most circumstances.
- = 0 Flush the driver and instruction pipelines only (outputs will be reset to format default)
- = 1 Flush the driver and instruction pipelines then load and advance the first vector to the outputs

EndMode Specifies the state of the Timing Generators after an END instruction.
- = 0 Stop On End.
- = 1 Lock On End.
- = 2 Cycle On End (An unconditional END will stop immediately.)

EPArmDuringLock

This functional call is the final EPIO function necessary before starting a new test with the Timing Generators "Locked" from a prior test. Previously, a test was executed having been armed with the **EPArm** functional call. In that call, the *EndMode* parameter has been set to "Lock On End." Now, whenever a test reaches an END instruction, the Timing Generators continue to operate, continually driving the last vector with Timing Set to the DUT. This allows the vector and fail address memory to be accessed while the product is kept active.

This functional call must be called once for every board in the system, as it enables the appropriate arm signals and loads the memory address register for each board. Since the *MasterEnable* parameter enables one board to drive several control signals onto a bus, the board that will be master should be armed last to avoid bus contention. The same master board used in previous tests must be used here.

The master board will be generating Timing Generator and Test Counter Enable gates between selected Start and Stop signals. The circuit is enabled for output and armed for the Start trigger specified by the **EPVectorGate**. JUMP and WAIT commands are also generated here, and the Fail flag is reset. This is usually the appropriate board to use as the argument for the **EPStatus** functional call.

This functional call may be called multiple times to re-arm boards for several consecutive tests. Since the *EndMode* cannot be changed by this functional call, after the final test, the **EPHalt** functional call must be used to shut down the Timing Generators.

Visual Basic Declaration: **epio32.bas**

Public Sub EPArmDuringLock (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal MasterEnable As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPArmDuringLock (int16 boardNumber, int32 startAddress, int16 masterEnable);

Call EPArmDuringLock(BoardNumber, StartAddress, MasterEnable)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid memory location to set the address counter to. It should be the location of the first vector in the test to be started.

MasterEnable Signifies if this board is controlling the execution flow.
= 0 This board will be a slave board.
= 1 This board will provide the command processing.

EPCarryMatrixSetup

This functional call is used to configure the algorithmic units available for Driver and Expected data. Algorithmic units are byte-wide and divided into four separate units per board. The algorithmic units may act as accumulators or they may simply latch vector data.

In the default mode of operation, the accumulators latch incoming data from the Driver/Expected memory. If secondary Add instructions are programmed using the **EPPutInstructionData** functional call, the incoming data will be added to the previously latched values. There is separate control over each accumulator on the board.

Each accumulator can select a carry-in signal for its Add operations using the *CarryInSourc*e parameter. This signal can be provided by a carry-out from one of the other three accumulators on the same board or from one of four global carry lines which can connect several EPIO boards together. Using these global carry lines, much larger accumulators may be constructed that would not have been otherwise possible with a single board.

Each accumulator's carry-out lines are hard-wired on the board and are always available as carry-ins to the other three local accumulators. Alternatively, the carry-out of any accumulator may be connected to one of the global carry lines using the *CarryOutDestination* parameter.

NOTE: Due to excessively long propagation delays, the maximum test speed will be reduced when using long carry logic chains.

Visual Basic Declaration: **epio32.bas**

Public Sub EPCarryMatrixSetup (ByVal BoardNumber As Integer, ByVal ByteNumber As Integer, ByVal CarryInSource As Integer, ByVal CarryOutDestination As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPCarryMatrixSetup (int16 boardNumber, int16 byteNumber, int16 carryInSource, int16 carryOutDestination);

Call EPCarryMatrixSetup(BoardNumber, ByteNumber, CarryInSource, CarryOutDestination)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

ByteNumber The byte number on the specified EPIO board.
= 0 Byte #0.
= 1 Byte #1.
= 2 Byte #2.
= 3 Byte #3.

CarryInSource This parameter selects the source of the carry-in signal.
= -1 Disable the carry-in line.
= 0 From byte #0.
= 1 From byte #1.
= 2 From byte #2.
= 3 From byte #3.
= 4 From global line #0.
= 5 From global line #1.
= 6 From global line #2.
= 7 From global line #3.

CarryOutDestination This parameter selects an off-board line to attach the carry-out signal to.
= -1 Disable global carry-out drivers from this byte.
= 0 To global line #0.
= 1 To global line #1.
= 2 To global line #2.
= 3 To global line #3.

EPChannelSetup

This functional call is used to configure EPIO board channels as drivers, receivers, or bi-directional channels. Each bit in two separate byte-wide parameters controls four channels on single board. A high logic value in any bit of the *DriverEnable* parameter configures four channels as drivers. A high logic value in any bit of the *ReceiverEnable* parameter configures four channels as receivers. Channels that have high logic values in the same position of both parameters will configure four channels as bi-directional.

The last four channels on each board (channels 28-31) control the directions of configured bi-directional channels in groups of eight. They cannot be configured as bi-directional channels; they must be full-time drivers or receivers. In addition, if any other channels on a board are bi-directional, then these four channels must be configured as full-time drivers. The call will return errors for violating these two rules.

Board	Enable Bit	Channels	Bi-directional Control Channel
0	0	0 - 3	28
0	1	4 - 7	28
0	2	8 - 11	29
0	3	12 - 15	29
0	4	16 - 19	30
0	5	20 - 23	30
0	6	24 - 27	31
0	7	28 - 31	Not Used
1	0	32 - 35	60
1	1	36 - 39	60
1	2	40 - 43	61
1	3	Etc.	

During execution, a high logic bit on any control channel will cause a corresponding group of bi-directional channels to drive out to the LFA. (and cause the LFA to drive to the product, if configured correctly). A logic low will disable the drivers, thus allowing data to be received from the LFA. This ability assumes the control channels are configured with the R0 output format, and allows the EPIO to emulate the RZ output format.

Unspecified channels will be configured as receivers by default. The user is responsible for masking these signals to prevent unwanted results.

Visual Basic Declaration: **epio32.bas**

Public Sub EPChannelSetup (ByVal BoardNumber As Integer, ByVal DriverEnable As Byte, ByVal ReceiverEnable As Byte)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPChannelSetup (int16 boardNumber, unsigned char driverEnable, unsigned char receiverEnable);

EPChannelSetup (BoardNumber, DriverEnable, ReceiverEnable)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

DriverEnable 8-bit number where each high bit represents a group of four channels configured as drivers.
= &H00 to &HFF.

ReceiverEnable 8-bit number where each high bit represents a group of four channels configured as receivers.
= &H00 to &HFF.

EPDisableBoard

This functional call is used to disable an EPIO board from driving out to the bus that connects the EPIO boards together. EPIO boards that are connected together are considered to be in a group. If a board is in a group but is not being used for a test, it is possible that the unused board can drive signals out to the Interconnect bus and cause problems with the boards actually being used for the test. Therefore, this functional call must be made for every board in a group that is not being used.

Note that this functional call does not need to be made if the **EPSetupConfig** functional call is used. The **EPSetupConfig** functional call will disable all boards in the same group as the master.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPDisableBoard (ByVal BoardNumber As Integer)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPDisableBoard (int16 boardNumber);
```

Call EPDisableBoard (BoardNumber)

WHERE:

BoardNumber Specifies the EPIO board to be disabled.

= 0 EPIO board 0.

= 1 EPIO board 1.

Etc. to board 7.

EPDriverFormat

This functional call is used to configure the output format for the drivers. This function is effective on fixed drivers or bi-directional channels; it will not have any effect on full-time receiver channels. Each individual channel is fully configurable for one of four different output formats and their complements. For examples of the different formats, please refer to Appendix B.

Visual Basic Declaration: **epio32.bas**

Public Sub EPDriverFormat (ByVal BoardNumber As Integer, ByVal Channel As Integer, ByVal OutputFormat As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPDriverFormat (int16 boardNumber, int16 channel, int16 outputFormat);

Call EPDriverFormat(BoardNumber, Channel, OutputFormat)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

Channel Channel offset on the specified board.

= 0 Channel 0.
= 1 Channel 1.
Etc. to channel 31.

OutputFormat Output format of the specified channel.

= 0 R0: Return-to-Zero
= 1 R1: Return-to-One
= 2 RC: Return-to-Complement
= 3 NR: Non-Return-to-Zero
= 4 /R0: Inverse Return-to-Zero
= 5 /R1: Inverse Return-to-One
= 6 /RC: Inverse Return-to-Complement
= 7 /NR: Inverse Non-Return-to-Zero

EPECTimingData

This functional call loads the Timing Generator memory on the Execution Controller. Similar to **EPIOTimingData**, it loads timing segment information into the Execution Controller, instead of the I/O Formatters.

Timing sets are defined as a collection of segments. The active and inactive regions of the timing sets may change only at the beginning of a segment. Since segments are defined for all four Trigger Matrix outputs at once, a segment must end whenever any of the four output triggers changes states. For the Trigger Matrix utility outputs, a trigger is created whenever there is an inactive-to-active transition (low-to-high) on the timing set, coinciding with a high value in the vector memory (shared with the Jump Memory; see **EPPutInstructionData**). This call must be used for every board even if the Trigger Matrix abilities are not used.

Individual timing sets are terminated by an end flag. As a rule, there must be at least three segments defined for any timing set. On the third segment of a set, or any thereafter, there may be a corresponding end flag placed. At the end of that segment, the vector cycle will end, and a new timing set will be loaded. Because of hardware restrictions, timing sets may only start on memory offsets evenly divisible by four. Also, timing sets must be the same length for the Execution Controllers and I/O Formatters on all boards. They are not required to have the same number of segments, but they must have the same total number of counts broken into at least three segments. See Appendix C for a complete list of rules.

For each vector cycle, the timing set is specified by the timing set selection Memory, which is loaded using the **EPPutTimingSelectData** functional call.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPECTimingData (ByVal BoardNumber As Integer, ByVal Length As Long,
ByRef SegmentLength() As Long, ByRef EndSegmentFlag() As Byte, ByRef
TMSegmentData() As Byte)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPECTimingData (int16 boardNumber, int32 length, u_int32
*segmentLength, u_char *endSegmentFlag, u_char *TMSegmentData);
```

Call EPECTimingData(BoardNumber, Length, SegmentLength, EndSegmentFlag, TMSegmentData)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

Length The number of steps in each array to write to board memory. Each step represents a timing segment. If the arrays do not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned.
= 0 to 1024.

SegmentLength() Array holding the length of each segment for the Execution Controller. Segments are used to define the length of a vector cycle and to position Trigger Matrix strobes. The length of any segment may be between 1 and 32768.

EndSegmentFlag() Array holding the end segment indicators. A high value in bit 0 indicates that the corresponding segment is the last in the timing set. A new vector will start after the segment is complete. One flag must exist for each Timing Set defined.

TMSegmentData() Array of numbers in which the first four bits represent an active or inactive segment of a Timing Set. The LSB of the number corresponds to the first Trigger Matrix utility channel.

EPEdgeClock

This functional call sets up the Edge Clock distribution hardware on the selected EPIO board. Edge clocks are responsible for timing edge placement and system synchronization. Only the master board's Edge Clock generator needs to be programmed. It will be distributed to the other boards in the same group.

The on-board DDS is the primary clock source. Its frequency is programmable over a large range of values. Aside from the preferred on-board DDS source, a software strobe may also be used (provided by the **EPStrobe** functional call). This is available for use by debug and utility software.

NOTE: By default, the power-on state of every EPIO is that of a slave board. To set up the Edge Clock, only one board needs to be called.

Visual Basic Declaration: **epio32.bas**

Public Sub EPEdgeClock (ByVal BoardNumber As Integer, ByVal ClockSource As Integer, ByVal Frequency As Double)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPEdgeClock (int16 boardNumber, int16 clockSource, double frequency);

Call EPEdgeClock(BoardNumber, ClockSource, Frequency)

WHERE:

BoardNumber Specifies the EPIO board number to be accessed.

=	0	EPIO board 0.
=	1	EPIO board 1.
		Etc. to board 7.

ClockSource Specifies the source of the edge clock.

=	-1	None (slave board does not generate an edge clock).
=	0	On-Board DDS.
=	3	Manual strobe (**EPStrobe**).

Frequency The frequency (in hertz) to set the board's edge clock generator to (DDS only).

300 to 70,000,000 Hz. (**Preliminary Specification**)

EPEdgeStart

This functional call is used to manually start the Timing Generators and is intended to be used with the master board. Once all boards are armed, this function will start the Timing Generators before the test begins. This is intended primarily for those cases where it may be useful to have a constant clock or pattern at the DUT before it is powered up. However, other uses may apply.

Visual Basic Declaration: **epio32.bas**

Public Sub EPEdgeStart (ByVal BoardNumber As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPEdgeStart (int16 boardNumber);

Call EPEdgeStart(BoardNumber)

WHERE:

BoardNumber Specifies the EPIO board to be accessed. This board must be the armed master.

= 0 EPIO board 0.

= 1 EPIO board 1.

Etc. to board 7.

EPExternalEndSetup

This functional call is used to configure the External Input Matrix for the END qualifiers used by the End-On-Qualifier instructions. The two instructions are EQ1 and EQ2. The Matrix allows one of eight receiver channels (16 - 23) to be routed to either of the two END qualifiers.

Visual Basic Declaration: **epio32.bas**

Public Sub EPExternalEndSetup (ByVal BoardNumber As Integer, ByVal EndQual1 As Integer, ByVal EndQual1Pol As Integer, ByVal EndQual2 As Integer, ByVal EndQual2Pol As Integer)

CVI Declaration: **dliepio.h**

u_int32 EPExternalEndSetup (int16 boardNumber, int16 endQual1, int16 endQual1Pol, int16 endQual2, int16 endQual2Pol);

Call EPExternalEndSetup (BoardNumber, EndQual1, EndQual1Pol, EndQual2, EndQual2Pol)

WHERE:

BoardNumber The EPIO board to be accessed. This board must be the master board.

= 0 to 7.

EndQual1 This parameter specifies which channel on the master board should be used as the input for EQ1.

= 16 to 23.

EndQual1Pol This parameter specifies the polarity for the input signal for EQ1.

= 0 Active low.

= 1 Active high.

EndQual2 This parameter specifies which channel on the master board should be used as the input for EQ2.

= 16 to 23.

EndQual2Pol This parameter specifies the polarity for the input signal for EQ2.

= 0 Active low.

= 1 Active high.

EPExternalJumpSetup

This functional call is used to configure the External Input Matrix for the JUMP qualifiers used by the Jump-On-Qualifier instructions. The two instructions are JQ1 and JQ2. The Matrix allows one of eight receiver channels (16 - 23) to be routed to either of the two JUMP qualifiers.

Visual Basic Declaration: **epio32.bas**

Public Sub EPExternalJumpSetup (ByVal BoardNumber As Integer, ByVal JumpQual1 As Integer, ByVal JumpQual1Pol As Integer, ByVal JumpQual2 As Integer, ByVal JumpQual2Pol As Integer)

CVI Declaration: **dliepio.h**

u_int32 EPExternalJumpSetup (int16 boardNumber, int16 jumpQual1, int16 jumpQual1Pol, int16 jumpQual2, int16 jumpQual2Pol);

Call EPExternalJumpSetup (BoardNumber, JumpQual1, JumpQual1Pol, JumpQual2, JumpQual2Pol)

WHERE:

BoardNumber The EPIO board to be accessed. This board must be the master board.

= 0 to 7.

JumpQual1 This parameter specifies which channel on the master board should be used as the input for JQ1.

= 16 to 23.

JumpQual1Pol This parameter specifies the polarity for the input signal for JQ1.

= 0 Active low.

= 1 Active high.

JumpQual2 This parameter specifies which channel on the master board should be used as the input for JQ2.

= 16 to 23.

JumpQual2Pol This parameter specifies the polarity for the input signal for JQ2.

= 0 Active low.

= 1 Active high.

EPExternalQualPolarity

This functional call is used to configure the external qualifiers used by the JUMP, WAIT and END on qualifier instructions. Each of these instructions has two external qualifier signals, EQ1 and EQ2, that are affixed to specific receiver inputs on the master board. There are six qualifiers used with six instructions that are available simply by implementing the proper primary instruction.

For setting the polarity on the two external Start triggers, see the **EPVectorGate** functional call.

Visual Basic Declaration: **epio32.bas**

Public Sub EPExternalQualPolarity (ByVal BoardNumber As Integer, ByVal JumpPolarity As Integer, ByVal WaitPolarity As Integer, ByVal EndPolarity as Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPExternalQualPolarity (int16 boardNumber, int16 jumpPolarity, int16 waitPolarity, int16 endPolarity);

Call EPExternalQualPolarity(BoardNumber, JumpPolarity, WaitPolarity, EndPolarity)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

=	0	EPIO board 0.
=	1	EPIO board 1.
		Etc. to board 7.

JumpPolarity Specifies the polarities for the external Jump qualifiers

=	0	Both external qualifiers active low
=	1	External qualifier 1 active high, qualifier 2 active low
=	2	External qualifier 1 active low, qualifier 2 active high
=	3	Both external qualifiers active high

WaitPolarity Specifies the polarities for the external Wait qualifiers

=	0	Both external qualifiers active low
=	1	External qualifier 1 active high, qualifier 2 active low
=	2	External qualifier 1 active low, qualifier 2 active high
=	3	Both external qualifiers active high

EndPolarity		Specifies the polarities for the external End qualifiers
=	0	Both external qualifiers active low
=	1	External qualifier 1 active high, qualifier 2 active low
=	2	External qualifier 1 active low, qualifier 2 active high
=	3	Both external qualifiers active high

EPExternalStartSetup

This functional call is used to configure the External Input Matrix for the Start qualifier used by the START circuitry. The START qualifier can be used to start a burst from an external input. The Matrix allows one of eight receiver channels (16 - 23) to be routed to the START qualifier.

Visual Basic Declaration: **epio32.bas**

Public Sub EPExternalStartSetup (ByVal BoardNumber As Integer, ByVal StartQual As Integer, ByVal StartQualPol As Integer)

CVI Declaration: **dliepio.h**

u_int32 EPExternalStartSetup (int16 boardNumber, int16 startQual, int16 startQualPol);

Call EPExternalStartSetup (BoardNumber, StartQual, StartQualPol)

WHERE:

BoardNumber The EPIO board to be accessed. This board must be the master board.

= 0 to 7.

StartQual This parameter specifies which channel on the master board should be used for the START qualifier.

= 16 to 23.

StartQualPol This parameter specifies the polarity for the input signal for the START qualifier.

= 0 Active low.

= 1 Active high.

EPExternalWaitSetup

This functional call is used to configure the External Input Matrix for the WAIT qualifiers used by the Wait-On-Qualifier instructions. The two instructions are WQ1 and WQ2. The Matrix allows one of eight receiver channels (16 - 23) to be routed to either of the two WAIT qualifiers.

Visual Basic Declaration: **epio32.bas**

Public Sub EPExternalWaitSetup (ByVal BoardNumber As Integer, ByVal WaitQual1 As Integer, ByVal WaitQual1Pol As Integer, ByVal WaitQual2 As Integer, ByVal WaitQual2Pol As Integer)

CVI Declaration: **dliepio.h**

u_int32 EPExternalWaitSetup (int16 boardNumber, int16 waitQual1, int16 waitQual1Pol, int16 waitQual2, int16 waitQual2Pol);

Call EPExternalWaitSetup (BoardNumber, WaitQual1, WaitQual1Pol, WaitQual2, WaitQual2Pol)

WHERE:

BoardNumber The EPIO board to be accessed. This board must be the master board.

= 0 to 7.

WaitQual1 This parameter specifies which channel on the master board should be used as the input for WQ1.

= 16 to 23.

WaitQual1Pol This parameter specifies the polarity for the input signal for WQ1.

= 0 Active low.

= 1 Active high.

WaitQual2 This parameter specifies which channel on the master board should be used as the input for WQ2.

= 16 to 23.

WaitQual2Pol This parameter specifies the polarity for the input signal for WQ2.

= 0 Active low.

= 1 Active high.

EPFailAddresses

This functional call retrieves the captured failed step addresses from the specified EPIO board. The desired number of failures to read is passed in through the *FailureCount* argument as well as an array that has been dimensioned to be at least that size. The actual number of failures is returned along with the fail addresses.

Step addresses are saved for every vector step where the Equal Flag is set to false while the Enable Fail Checking secondary instruction is asserted. The global Equal Flag is an open-collector product of all unmasked receiver bits compared to Expected data.

Because there may be execution loops and jumps in the test, the same fail address may be listed more than once, or the addresses may appear out of order. Fail addresses are always saved in the order in which they occur during a test.

Visual Basic Declaration: **epio32.bas**

Public Sub EPFailAddresses (ByVal BoardNumber As Integer, ByRef FailureCount As Long, ByRef FailureAddresses() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPFailAddresses (int16 boardNumber, int32 *failureCount, int32 *failureAddresses);

Call EPFailAddresses(BoardNumber, FailureCount, FailureAddresses())

WHERE:

BoardNumber Specifies the EPIO board to be accessed. This board must be the master board.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

FailureCount Passed in - the maximum number of elements to be returned in the FailAddresses array.
Returned - the actual number of failures downloaded in the array.

= 0 to 255 (limit of hardware).

FailureAddresses() Array of 19 bit numbers read from the EPIO board's Fail Address memory. Each address is a failed vector in the test.

EPGetDriverData

This functional call will download the stored Driver data from a specified EPIO board. The starting address in memory is given as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutDriverData**.

The Driver Memory is physically shared with the Expected Memory; therefore, this call is identical to **EPGetExpectedData**. The data must be combined before loading with either one of these functional calls. This will not present a conflict because a channel is exclusively either a driver or a receiver for any given vector cycle; therefore, only one of the calls needs to be used.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetDriverData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length as Long, ByRef DriverData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetDriverData (int16 boardNumber, int32 startAddress, int32 length, int32 *driverData);

Call EPGetDriverData (BoardNumber, StartAddress, Length, DriverData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data array from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

DriverData() The array of 32-bit numbers to be filled with the downloaded data.

EPGetExpectedData

This functional call will download the stored Expected data from a specified EPIO board. The starting address in memory is given as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutExpectedData**.

The Driver Memory is physically shared with the Expected Memory; therefore, this call is identical to **EPGetDriverData**. The data must be combined before loading with either one of these functional calls. This will not present a conflict because a channel is exclusively either a driver or a receiver for any given vector cycle; therefore, only one of the calls needs to be used.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetExpectedData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ExpectedData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetExpectedData (int16 boardNumber, int32 startAddress, int32 length, int32 *expectedData);

Call EPGetExpectedData(BoardNumber, StartAddress, Length, ExpectedData())

WHERE:

BoardNumber		Specifies the EPIO board to be accessed.
=	0	EPIO board 0.
=	1	EPIO board 1.
		Etc. to board 7.
StartAddress		Any valid address to retrieve the data from.
Length		The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.
ExpectedData()		The array of 32-bit numbers to be filled with the downloaded data.

EPGetExpectedNMLData

This functional call will download the stored Expected NML data from a specified EPIO board. The starting address in memory is given as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutExpectedNMLData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetExpectedNMLData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ExpectedNMLData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetExpectedNMLData (int16 boardNumber, int32 startAddress, int32 length, int32 *expectedNMLData);

Call EPGetExpectedNMLData(BoardNumber, StartAddress, Length, ExpectedNMLData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ExpectedNMLData() The array of 32-bit numbers to be filled with the downloaded data.

EPInhibitFail

This functional call is used to inhibit a channel from causing a failure without having to modify the expected or mask data. The 32-bit parameter *InhibitBits* is used to specify which channel(s) should be inhibited. A logic one in *InhibitBits* means the corresponding channel should be inhibited from causing a failure. For example, if *InhibitBits* is &H00010002, channel numbers 1 and 16 will be inhibited from causing a failure.

This ability is useful during the debug phase of generating a test. One or more channels can be inhibited from causing a failure if the user does not care about failures caused by that channel. Thus, only the channels they are currently concerned about will be flagged as failures.

Visual Basic Declaration: **epio32.bas**

Public Sub EPInhibitFail (ByVal BoardNumber As Integer, ByVal InhibitBits As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPInhibitFail(int16 boardnumber, u_int32 InhibitBits);

EPInhibitFail (BoardNumber, InhibitBits)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.

= 1 EPIO board 1.

Etc. to board 7.

InhibitBits 32-bit number where each bit represents one of the 32 channels on a board. If a bit is high, the corresponding channel is inhibited from causing a failure. If low, a channel will cause a failure if the received data is not equal to the expected data.

= &H00000000 to &HFFFFFFFF

EPGetInstructionData

This functional call will download the stored Instruction and Jump data from a specified EPIO board. The starting address in memory is given as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutInstructionData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPInstructionData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef InstructionData() As Integer, ByRef JumpData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetInstructionData (int16 boardNumber, int32 startAddress, int32 length, int32 *instructionData, int32 *jumpData);

Call EPGetInstructionData(BoardNumber, StartAddress, Length, InstructionData, JumpData)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the arrays do not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address.

InstructionData() An array of 32-bit numbers to be filled with the downloaded Instruction data.

JumpData() An array of 32-bit numbers to be filled with the downloaded Jump data.

EPGetLabelStep

This functional call is used to return the step number (or vector number) where the requested label is located. Labels are assigned to step numbers in the EPIO Pattern Editor program. Within the editor program, labels have two functions. One is as the operand for a jump instruction and the second is to mark the location of a vector. By using labels instead of absolute numerical addresses, vectors can be inserted or removed without having to go back through all of the vectors and modifying the absolute operand(s) of any affected instructions. This call is used to retrieve the absolute numerical address associated with a label.

It should be noted that before executing this call, the **EPLoadVector** call must have been made. This is required because the labels and their locations are contained in the vector file. Therefore, when **EPLoadVector** is used, the labels and their location are placed into a table, which is then accessed by this function to call the location of the label.

If a '-1' is returned as the *StepNumber*, it means the requested *Label* could not be found.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPGetLabelStep(ByVal Label As String, ByRef StepNumber As Long)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPGetLabelStep (char *label, int32 *stepNumber);
```

Call EPGetLabelStep(Label, StepNumber)

WHERE:

Label The label to search for.

StepNumber The step number associated with the given label.

EPGetMaskData

This functional call will download the stored Mask data from a specified EPIO board. The starting address in memory is given, as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutMaskData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetMaskData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef MaskData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetMaskData (int16 boardNumber, int32 startAddress, int32 length, int32 *maskData);

Call EPGetMaskData(BoardNumber, StartAddress, Length, MaskData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

MaskData() The array of 32-bit numbers to be filled with the downloaded data.

EPGetResultData

This functional call will download the Result data from the specified EPIO board. The starting address in memory is given as well as the number of values to read. This data is the actual received data prior to any masking.

Because there may be execution loops in the test, areas of this memory may be overwritten several times during test execution. Data that may have been previously stored will be lost. The Fail Flag and Fail Address Register are provided to determine the failures that an execution may have had. The Result DataMemory is intended mainly for post-test analysis of failures.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetResultData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ResultData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetResultData (int16 boardNumber, int32 startAddress, int32 length, int32 *resultData);

Call EPGetResultData(BoardNumber, StartAddress, Length, ResultData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ResultData() An array of 32-bit numbers read from the EPIO board Result Memory. The LSB of the number corrresponds to the first channel on the board.

EPGetResultNMLData

This functional call will download the result no-man's-land (NML) data from the specified EPIO board. The starting address in memory is given as well as the number of values to read. This data is the actual received NML data.

Because there may be execution loops in the test, areas of this memory may be overwritten several times during test execution. Data that may have been previously stored will be lost. The Fail Flag and Fail Address Register are provided to determine the failures that an execution may have had. The Result NML Data Memory is intended mainly for post-test analysis of failures.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetResultNMLData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ResultNMLData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetResultNMLData (int16 boardNumber, int32 startAddress, int32 length, int32 *resultNMLData);

Call EPGetResultData(BoardNumber, StartAddress, Length, ResultNMLData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ResultNMLData() An array of 32-bit numbers read from the EPIO board Result NMLMemory. The LSB of the number corrresponds to the first channel on the board.

EPGetTimingSelectData

This functional call will download the stored Timing Set Selection data from a specified EPIO board. The starting address in memory is given as well as the number of values to load. This call is provided as a means to investigate the contents previously uploaded with **EPPutTimingSelectData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPGetTimingSelectData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef TimingSelectData() As Byte)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPGetTimingSelectData (int16 boardNumber, int32 startAddress, int32 length, u_char *timingSelectData);

Call EPGetTimingSelectData(BoardNumber, StartAddress, Length, TimingSelectData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to retrieve the data from.

Length The number of steps to retrieve from board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

TimingSelectData() An array of 32-bit numbers to be filled with the downloaded data.

EPHalt

This functional call generates a stop signal and halts the edge clocks in the system. Since the actual stopping address cannot be accurately predicted ahead of time, it is not meant to be used to pause the test. Its primary use is to stop the Timing Generators after a “Lock on End” condition has been reached after executing a test or a chain of tests.

It may also be used stop the currently running test during the debugging phase of a project. A timeout could also be manufactured in the test code to prevent a product test from “locking up.” This call only has a practical effect on the master board.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPHalt (ByVal BoardNumber As Integer)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPHalt (int16 boardNumber);
```

Call EPHalt(BoardNumber)

WHERE:

BoardNumber Specifies the EPIO board to be accessed. This must be the master board.

= 0 EPIO board 0.

= 1 EPIO board 1.

Etc. to board 7.

EPIOTimingData

This functional call programs the Timing Generators on the I/O Formatters. Similar to **EPECTimingData**, it programs timing segment information on the I/O Formatters instead of the Execution Controller.

Timing sets are defined as a collection of segments. The active and inactive regions of the timing sets may change only at the beginning of a segment. Since segments are defined for all eight channels at once, a segment must end whenever any of the eight outputs change state or any of the eight receivers strobe. For the receivers, a strobe is created whenever there is a high-to-low transition of the timing set. Behavior at the driver outputs depends on the format selected by the **EPDriverFormat** functional call.

Individual timing sets are terminated by an end flag. As a rule, there must be at least three segments defined for any timing set. On the third segment of a set or any thereafter, there may be an end flag placed. At the end of that segment, the vector will end, and a new timing set will be loaded. Because of hardware restrictions, timing sets may only start on memory offsets evenly divisible by four. Also, timing sets must be the same length for the Execution Controllers and I/O Formatters on all boards. They are not required to have the same number of segments but they must have the same total number of counts broken into at least three segments. See Appendix C for a complete list of rules.

For each vector cycle, the timing set is specified by the Timing Set Selection Memory, which is loaded using the **EPPutTimingSelectData** functional call.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPIOTimingData (ByVal BoardNumber As Integer, ByVal ByteNumber As
Integer, ByVal Length As Long, ByRef SegmentLength() As Long, ByRef
EndSegmentFlag () as Byte, ByRef DrvrSegmentData() As Byte, ByRef
RcvrSegmentData() As Byte)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPIOTimingData (int16 boardNumber, int16 byteNumber, int32
length, u_int32 *segmentLength, u_char *endSegmentFlag, u_char *drvrSegmentData,
u_char *rcvrSegmentData);
```

Call EPIOTimingData(BoardNumber, ByteNumber, Length, SegmentLength(), EndSegmentFlag(), DrvrSegmentData(), RcvrSegmentData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

ByteNumber The byte number on the board to be loaded with its Driver and Receiver Timing Set information.
= 0 to 3.

Length The number of steps from each of the arrays to write to board memory. Each step represents a timing segment. If the arrays do not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned.
= 0 to 1024.

SegmentLength() Array holding the length of each segment for the I/O Formatter. Segments are used to define the length of a vector cycle and to position receiver strobes and driver active regions. The length of any segment may be between 1 and 32768.

EndSegmentFlag() Array holding the end segment indicators. A high value in bit 0 indicates that the corresponding segment is the last in the timing set. A new vector will start after the segment is complete. One flag must exist for each Timing Set defined.

DrvrSegmentData() Array of 8-bit numbers in which each bit represents an active (high) or inactive (low) segment phase of a driver channel's Timing Set. The LSB of the number corresponds to the first driver channel.

RcvrSegmentData() Array of 8-bit numbers in which each bit represents a high or low phase of a receiver channel's Timing Set. Receivers always strobe on a high-to-low transition. The LSB of the number corresponds to the first receiver channel.

EPLoadVector

This functional call is used to upload the test vectors (contained in the vector file created by the EPIO Pattern Editor program) into EPIO memory. The vector file contains all of the vectors for all of the EPIO boards used for one test. The memory location to load the vectors to is given by *LoadAddress*. This call is used in conjunction with **EPSetupConfig**. The **EPSetupConfig** functional call is used to configure the EPIO boards for the test and must be used first.

Visual Basic Declaration: **epio32.bas**

Public Sub EPLoadVector(ByVal VectorFileName As String, ByVal LoadAddress As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPLoadVector (char *vectorFileName, int32 loadAddress);

Call EPLoadVector(VectorFileName, LoadAddress)

WHERE:

VectorFileName Full path to the vector file created by the EPIO Editor program.

LoadAddress Any valid address to write the data to.

EPPutDriverData

This functional call will upload the Driver data onto a specified EPIO board. The starting address in memory is given as well as the number of values to load. The data specified will be output from the drivers during the active phase of the Timing Set. This data may be later retrieved with the **EPGetDriverData** functional call.

The Driver Memory is physically shared with the Expected Memory; therefore, this function is identical to **EPPutExpectedData**. The data must be combined before loading with either one of these functional calls. This will not present a conflict because a channel is exclusively either a driver or a receiver for any given vector cycle; therefore, it is necessary to use only one of these calls.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutDriverData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length as Long, ByRef DriverData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutDriverData (int16 boardNumber, int32 startAddress, int32 length, int32 *driverData);

Call EPPutDriverData (BoardNumber, StartAddress, Length, DriverData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address in board to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address.

DriverData() Array of 32-bit numbers in which each bit represents the output state of a single driver during the active phase of a given step. The LSB of the number corresponds to the first channel on the board. Expected data for receiver vectors must be included in this array.

EPPutExpectedData

This functional call will upload data comparison values onto the specified EPIO board. The starting address in memory is given as well as the number of values to load. By implementing an Exclusive-OR function, the data received will be compared to this Expected data and the results will be combined with all other bits for each vector step. These results set a global Equal Flag, which may be used as arguments for several test instructions, and a global Fail Flag. Any channels not required for production of the Equal Flag should be masked off using the Mask memory (programmed with **EPPutMaskData**.

The Expected Memory is physically shared with the Driver Memory; therefore, this call is identical to **EPPutDriverData**. The data must be combined before loading with either one of these functional calls. This will not present a conflict because a channel is exclusively either a driver or a receiver for any given vector cycle; therefore, it is necessary to use only one of these calls. This data may be later retrieved with the **EPGetExpectedData** functional call.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutExpectedData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ExpectedData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutExpectedData (int16 boardNumber, int32 startAddress, int32 length, int32 *expectedData);

Call EPPutExpectedData(BoardNumber, StartAddress, Length, ExpectedData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ExpectedData() Array of 32-bit numbers in which each bit represents the XOR mask of a single receiver at a given step. This data represents the expected state of every bit at any point in time. The LSB of the number corresponds to the first channel on the board. Driver data for driver states must be included in this array.

EPPutExpectedNMLData

This functional call will upload the no-man's-land (NML) comparison values onto the specified EPIO board. The starting address in memory is given as well as the number of values to load. By implementing an Exclusive-OR function, the data received will be compared to this Expected NML data and the results will be combined with all other bits for each vector step. These results set a global Equal Flag, which may be used as arguments for several test instructions, and a global Fail Flag. Any channels not required for production of the Equal Flag should be masked off using the Mask Memory (programmed with the **EPPutMaskData** functional call).

This data may be later retrieved with **EPGetExpectedNMLData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutExpectedNMLData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ExpectedNMLData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutExpectedNMLData (int16 boardNumber, int32 startAddress, int32 length, int32 *expectedNMLData);

Call EPPutExpectedNMLData(BoardNumber, StartAddress, Length, ExpectedNMLData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ExpectedNMLData() Array of 32-bit numbers in which each bit represents the XOR mask of a single receiver at a given step. This data represents the expected state of every bit at any point in time. The LSB of the number corresponds to the first channel on the board.

EPPutInstructionData

This functional call will upload the instruction and jump tables for the specified EPIO board. The starting address in memory is given as well as the number of values to load. The EPIO instruction set consists of several primary and secondary instructions. Only one primary instruction may be processed in one vector cycle; however, multiple secondary instructions may be processed in the same cycle.

The instruction of each vector is 16 bits long. Primary instructions are decoded from the least significant five bits, and the remaining 11 bits contain secondary instructions. Four additional instruction bits are "borrowed" from the jump memory. The most significant 4 bits of the jump table are always used for the Trigger Matrix Utility strobes unless a Utility Counter Load instruction is also given. Then the 4 bits are a part of the counter value to load.

Other than the Trigger Matrix instructions, the jump table also contains the Utility Counter load values (24 bits) and the Jump addresses (20 bits). This means that the JUMP and 'Load Counter' instructions are also mutually exclusive on any given vector cycle despite the fact that one is a primary instruction while the other is a secondary instruction.

The same jump table must be uploaded onto every EPIO board used in a test since each board loads and increments its own vector address counter. An instruction table should also be uploaded onto every EPIO board containing the instructions for the algorithmic units on that board.

Use the **EPExternalQualPolarity** functional call to set the polarity on the external qualifiers for the JUMP, WAIT, and END instructions.

This data may be later retrieved with **EPGetInstructionData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutInstructionData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef InstructionData() As Integer, ByRef JumpData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutInstructionData (int16 boardNumber, int32 startAddress, int32 length, int32 *instructionData, int32 *jumpData);

Primary Instructions (Bits 0-4, only useful on master board):

Binary Code	Instruction
00000	No Operation
00001	End (Unconditional)
00010	End On Unequal
00011	End On Equal
00100	Wait On Unequal
00101	Wait On Equal
00110	Wait On Qualifier 1
00111	Wait On Qualifier 2
01000	Delay On Counter 1
01001	Delay On Counter 2
01010 - 01111	No Operation (Reserved)
10000	Jump (Unconditional)
10001	Jump On Unequal
10010	Jump On Equal
10011	Jump On Carry Clear Counter 1
10100	Jump On Carry Clear Counter 2
10101	Jump On Qualifier 1
10110	Jump On Qualifier 2
10111 - 11111	No Operation (Reserved)

Secondary Instructions

Bit(s)	Instruction	States	Master
5, 6	Spares		
7	Enable Fail Checking	0 - Disable, 1 - Enable	Y
8 - 9	Utility Counter 1 Control	00 - Hold, 01 - Up, 10 - Down, 11 - Load	Y
10 - 11	Utility Counter 2 Control	00 - Hold, 01 - Up, 10 - Down, 11 - Load	Y
12 - 15	Accumulator Control Byte 0 - 3	0 - Latch mode, 1 - Add	
20 - 23 (Jump Table Overlap)	Trigger Matrix Utility Outputs	Rising edge trigger, timing sets applied	

Note that Utility Counters may not be loaded during any JUMP instructions because the jump table is shared between the two instructions. (JUMP instructions take precedence.)

Also note that Trigger Matrix Utility strobes may not occur during Utility Counter loads because the jump table is shared between the two instructions. (Counter loads take precedence).

Call EPPutInstructionData(BoardNumber, StartAddress, Length, InstructionData(), JumpData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in each array to write to board memory. If the arrays do not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

InstructionData() An array of 16-bit encoded instructions.

JumpData() An array of 24-bit numbers to be used as arguments to corresponding instructions. (Also contains Trigger Matrix Instructions.)

EPPutMaskData

This functional call will load data for the evaluation Mask Memory onto the specified EPIO board. The starting address in memory is given as well as the number of values to load. By implementing an AND function, the result of the Expected data and NML comparisons will be ignored for purposes of evaluation. The result bits are used for comparison purposes for several test instructions and to create a global Fail Flag.

This data may be later retrieved with **EPGetMaskData**.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutMaskData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef MaskData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutMaskData (int16 boardNumber, int32 startAddress, int32 length, int32 *maskData);

Call EPPutMaskData(BoardNumber, StartAddress, Length, MaskData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write data to.

Length The number of steps in the array to write to board memory. If the arrays do not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

MaskData() An array of 32-bit numbers in which each bit represents the AND mask of a single receiver at a given step. The LSB of the number corresponds to the first channel on the board.

EPPutResultData

This functional call uploads data to the Result Data Memory on the specified EPIO board. The starting address in memory is given as well as the number of values to read.

There is very little practical use to this functional call other than setting the memory to a known state before a test is run. Use the **EPGetResultData** functional call to retrieve test results.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutResultData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ResultData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutResultData (int16 boardNumber, int32 startAddress, int32 length, int32 *resultData);

Call EPPutResultData(BoardNumber, StartAddress, Length, ResultData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address can be used.

ResultData() An array of 32-bit numbers to be loaded into the EPIO board Result Data Memory. The LSB of the number corresponds to the first channel on the board.

EPPutResultNMLData

This functional call uploads data to the "no-man's-land" Result NML Data Memory on the specified EPIO board. The starting address in memory is given as well as the number of values to read.

There is very little practical use to this functional call other than setting the memory to a known state before a test is run. Use the **EPGetResultNMLData** functional call to retrieve NML test results.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutResultNMLData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef ResultNMLData() As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutResultNMLData (int16 boardNumber, int32 startAddress, int32 length, int32 *resultNMLData);

Call EPPutResultNMLData(BoardNumber, StartAddress, Length, ResultNMLData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address.

ResultNMLData() An array of 32-bit numbers to be uploaded into the EPIO board Result NML Data Memory. The LSB of the number corresponds to the first channel on the board.

EPPutTimingSelectData

This functional call will upload Timing Set Selection data onto the specified EPIO board. The starting address in memory is given as well as the number of values to load. The values stored are the eight most significant bits of the address of the Timing Set to use for each corresponding vector cycle in the loaded test. Timing Set Memory is 1024 elements (segments) thereby requiring 10 bits to address all of the cells. However, the hardware restricts each Timing Set starting point to element numbers divisible by four within the Timing Set Memory. Thus, only eight bits of address information is required.

The Timing Sets themselves are loaded with the **EPIOTimingData** and **EPECTimingData** functional calls. This data may be later retrieved with the **EPGetTimingSelectData** functional call.

Visual Basic Declaration: **epio32.bas**

Public Sub EPPutTimingSelectData (ByVal BoardNumber As Integer, ByVal StartAddress As Long, ByVal Length As Long, ByRef TimingSelectData() As Byte)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPPutTimingSelectData (int16 boardNumber, int32 startAddress, int32 length, u_char *timingSelectData);

Call EPPutTimingSelectData(BoardNumber, StartAddress, Length, TimingSelectData())

WHERE:

BoardNumber Specifies the EPIO board to be accessed.
= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartAddress Any valid address to write the data to.

Length The number of steps in the array to write to board memory. If the array does not contain the specified number of steps, an error is returned. If the physical memory remaining is not large enough, an error is returned. Any value that is less than the total memory size minus the starting address.

TimingSelectData() An array of 8-bit numbers that specifies the starting location of the timing set to use for each corresponding vector.

EPSetupConfig

This functional call is used to configure the EPIO boards used for one test. The information used to configure the boards is contained in the file given by *ConfigFileName*. This file is created by the EPIO Pattern Editor program. All of the boards associated with one test are configured with this call. This call is used in conjunction with the **EPLoadVector** functional call which loads all of the test vectors into the EPIO memory. It should be noted that this call does not configure the Logic Family Adapter boards (LFA). The LFAs must be configured separately.

Visual Basic Declaration: **epio32.bas**

```
Public Sub EPSetupConfig(ByVal ConfigFileName As String)
```

CVI Declaration: **dliepio.h**

```
u_int32 DLIepio_EPSetupConfig (char *configFileName);
```

Call EPSetupConfig(ConfigFileName)

WHERE:

ConfigFileName Full path to the configuration file created by the EPIO Editor program.

EPStart

This functional call is used to give a manual start signal to the EPIO master board. The start signal creates the Test Counter Enable signal that is distributed from the master to all EPIO boards in the group. Previously, the board needs to have been set up using the **EPVectorGate** and the **EPArm** or **EPArmDuringLock** functional calls.

Visual Basic Declaration: **epio32.bas**

Public Sub EPStart (ByVal BoardNumber As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPStart (int16 boardNumber);

Call EPStart(BoardNumber)

WHERE:

BoardNumber Specifies the EPIO board to be accessed. This board must be the armed master.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

EPStatus

This functional call retrieves a variety of useful information from the EPIO board. The status registers on the board includes the state of the Start, Stop, Equal, and Fail flags as well as the state of several logic products and signals. This information will be returned in the form of a single word of data with each bit representing one flag or signal. A table of explanation is provided on the next page.

The functional call may also return several other registers useful in determining the status or progress of a test. The vector address counter, utility counters and fail counter may all be read. Usually, the master board contains the most complete set of information; however any board may be selected.

Visual Basic Declaration: **epio32.bas**

Public Sub EPStatus (ByVal BoardNumber As Integer, ByVal RegisterOffset As Integer, ByRef Status As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPStatus (int16 boardNumber, int16 registerOffset, int32 *status);

Call EPStatus(BoardNumber, RegisterOffset, Status)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

RegisterOffset Selects the information to be read.

= 0 Status flags and signals. (See table on next page)
= 1 Vector Address Counter.
= 2 Utility Counter 1.
= 3 Utility Counter 2.
= 4 Fail Counter.

Status 32-bit return value.

EPStatus Bit Designations for RegisterOffset = 0

Bit	Function
0	Test Counter Start Signal
1	Test Counter Stop Signal
2	Test Counter Enable
3	Test Counter is running
4	Timing Generator Start Signal
5	Tining Generator Stop Signal
6	Timing Generator Enable
7	Timing Generator is running
8	FAIL Flag - At least one failure occurred during the test.
9	Fail Address buffer overflowed (> 255)
10	Wait Instruction is being executed
11	Cycle on End is being executed (EndMode)
12	Reserved
13	Reserved
14	Reserved
15	Reserved

Note: 0 = False, 1 = True

EPStrobe

This functional call is used to manually strobe the Edge Clock on the master EPIO board. This only has a practical effect on the board supplying the Master Edge Clock to the group. It is used to single step through each edge of a test program for debugging purposes. To enable this call, the **EPEdgeClock** functional call must have been configured to expect a software strobe.

The number of Edge Clocks to generate may be specified to easily advance one vector step or portion of a vector. However, the burst timing is not user selectable. The bursts will occur in rapid succession, but are not separated by a specific time interval.

Visual Basic Declaration: **epio32.bas**

Public Sub EPStrobe (ByVal BoardNumber As Integer, ByVal StrobeCount As Long)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPStrobe (int16 boardNumber, int32 strobeCount);

Call EPStrobe(BoardNumber, StrobeCount)

WHERE:

BoardNumber Specifies the EPIO board to be accessed. The board must be the master.

= 0 EPIO board 0.

= 1 EPIO board 1.

Etc. to board 7.

StrobeCount The number of edge clocks to generate.

= 1 to 1024 (segments) X 32768 (counts) = 33,554,432. (This is the theoretical maximum length of a single vector. The system is capable of processing up to 625,000 strobes per second.)

EPVectorGate

This functional call configures how the EPIO boards get started and stopped and can enable or disable the End-On-Fail mode. Mainly, it sets up the hardware that creates the Test Counter Enable signal. The Test Counter Enable signal is used to allow the advancement of the memory address counter, which indexes memory for pattern output, Expected data, instructions, and other storage needs. The Test Counter Enable exists between start and stop triggers specified by this functional call.

Several sources may initiate the start trigger. These include a software manual start using the **EPStart** functional call, Trigger Matrix input, or an external signal supplied by one of eight dedicated receivers. Eight dedicated receiver channels can be setup to supply the Start signal. However, only one of the receiver channels can be enabled at a time. The polarity is selectable on the externally supplied inputs by using **EPExternalStartSetup**. The Timing Generator Enable will also be started by the Start signal if they are not already running when the trigger occurs.

Several sources may end the gate, including a Trigger Matrix input or any "End" instructions programmed by the user (see the **EPPutInstructionData** functional call). The **EPHalt** call may also be used to abruptly end a test; however, its use is mostly intended for debug situations.

This functional call will also enable or disable the ability to stop on a failure. If this mode is enabled, the test will stop if an unequal condition occurs and Fail Checking (a secondary instruction) is enabled. Please note that due to the hardware pipeline, the vector the test stops on will not be the vector with the unequal condition. After an End-On-Fail occurs, the vector being output to the Patchboard will be a few vectors after the unequal condition. The End-On-Fail mode should not be confused with the End-On-Fail instruction. The End-On-Fail mode will stop a test on the first unequal condition (if Fail Checking is enabled on that vector). The End-On-Fail instruction will only stop the test after the End-On-Fail instruction has been processed and an unequal condition previously occurred. In addition, when using the End-On-Fail instruction, a test will not stop if the unequal condition occurs on the same vector as the End-On-Fail instruction (this is due to the hardware pipeline).

The **EPArm** or **EPArmDuringLock** functional calls must be used to re-Arm the boards. For setting the polarity on the external Jump, Wait, and End qualifiers, see the **EPExternalQualPolarity** functional call.

Visual Basic Declaration: **epio32.bas**

Public Sub EPVectorGate (ByVal BoardNumber As Integer, ByVal StartSignal As Integer, ByVal StopSignal As Integer, ByVal EndOnFail As Integer)

CVI Declaration: **dliepio.h**

u_int32 DLIepio_EPVectorGate (int16 boardNumber, int16 STARTSignal, int16 STOPSignal, int16 endOnFail);

Call EPVectorGate(BoardNumber, StartSignal, StopSignal, EndOnFail)

WHERE:

BoardNumber Specifies the EPIO board to be accessed.

= 0 EPIO board 0.
= 1 EPIO board 1.
Etc. to board 7.

StartSignal Specifies the source of the START signal.

= 0 Strobe from the CPU using **EPStart**.
= 1 Signal from the Trigger Matrix.
= 2 Signal from the external START matrix.

StopSignal Enable/Disable the Trigger Matrix Stop signal ("End instructions are always enabled.)

= 0 Disable the Stop signal from the Trigger Matrix.
= 1 Enable the Stop signal from the Trigger Matrix.

EndOnFail Enable/Disable the ability to stop a test when a failure occurs. If this mode is enabled, the test will stop when an unequal condition. occurs and Fail Checking is enabled for the unequal vector.

= 0 Disable the ability to stop on a failure.
= 1 Enable the ability to stop on a failure.

Logic Family Adapters

Logic Family Adapters

Each EPIO Board is paired with a Logic Family Adapter (LFA). The purpose of the LFA is to connect the drive and receive channels of the EPIO Board to the Device-Under-Test (DUT), while keeping the distance between the LFA and DUT to a minimum. The distance between the EPIO Board and the LFA may be lengthened somewhat, as differential signaling maintains signal integrity.

This arrangement also allows the LFA to adapt the digital logic levels to the requirements of the DUT. The LFA is designed specifically for the targeted functional test(s) and user requirements. For instance, an adjustable-level LFA accommodates a wide range of logic levels, or a faster, fixed voltage LFA may be more desirable.

The LFA must be designed specifically to support whatever features are required by the test application. Some of the features outlined in the **Drivers** and **Receivers** sections of the EPIO Board, above, may only be used if the selected LFA supports them. Such features include on-the-fly bi-directional channels and no-man's-land detection (dual-level receiver thresholds.) Other features, such as programmable logic levels and disconnect relays, may also be added to the design of an LFA.

Note: It is important to remember that EPIO Board channel directions, configured with the **EPChannelSetup** functional call, must be mirrored correctly on the LFA as well. See the product documentation for the specific model of LFA used to find the appropriate method. Failure to do so will result in the test not performing as expected.

Selftest

EPIO Selftest Routines

The EPIO selftest routines are designed to fully test the functionality of the EPIO board. The EPIO Selftest board will be required for some of the EPIO's selftest routines and it must reside in the same slot in the Selftest fixture as the EPIO board is in the Testhead. The Analog Selftest Executive is used to execute the EPIO Selftest routines.

EPIO_Dig_f

This selftest routine checks the ability of the EPIO board to read and write to its own registers and checks the ability to load the address counter and its ability to count up to its maximum address. The register test is performed by doing a walking 1's test on each of the write/readable registers on the EPIO board. The values 0x00 and 0xFF are also used for a total of ten tests. For each test, a known pattern is written to the register. Then the inverted data is written to a different register. This is done to modify the data bus and prevent any capacitive loads from maintaining the original data. Finally the register that was written to is read and verified that the correct data was read.

The second test performs a walking ones test on the address counter. This is done by loading an address that has one bit high. This loaded address is then read back and verified that it is correct. This is done for all 24 bits that make up the address counter. The second part of the address counter test checks the ability of the counter to count. This is done by loading an address into the counter and then strobing the counter a set number of times. The address is then read and verified that it is at the correct location. The test is done this way for 16 blocks with each block being strobed 32K counts (for a total of 512K, the maximum depth of the EPIO memory).

A third test verifies that the TClear() functional call can clear the registers. The ability to clear the registers is needed to reset the board to a known state. For this test, each of the testable registers is filled with different data. Once they are all filled, they are read and it is verified that they contain the correct data. Then, TClear() is called to clear all of the registers. All of the registers are then verified that they are cleared.

This is the most basic of the EPIO selftest routines. If this test fails, any failures on the other selftests will be inconclusive. This is because, if the digital data bus has a problem, it is unknown what data was written to a register. On the other hand, if this test fails, some of the other selftests may pass depending on where

the digital failure is located and what is being testing by the selftest routine that passed. In addition, if the address counter test fails, any selftests that use the address counter would also likely fail. The Selftest fixture is not needed for this test.

Sample Output

```
-------------------------------------------------------------------------------------------
R 0X3       0       TST: 2          ADR:Ox4     EXP:Ox00      ACT:Ox00
RST D       0       TST: 25257      ADR:Ox3     EXP:Ox01      ACT:Ox01
RST CL      0       TST: 25307      ADR:Ox3     EXP:Ox00      ACT:Ox00
CNTREG      0       TST: 18493      ADR:Ox6D    EXP:Ox04      ACT:Ox04
COUNT       0       TST: 18513      ADR:Ox6D    EXP:Ox17FFF   ACT:Ox17FFF
-------------------------------------------------------------------------------------------
'R 0X3'         -> Testing the register at offset 0x03 on board zero
'ADR:0x4'       -> While testing offset 0x3, offset 0x4 was verified that it was cleared
‘RST D’         -> While testing the ability for TClear to clear the registers, this line verifies
                   that the test data was written to the register.
‘RST CL’        -> While testing the ability for TClear to clear the registers, this line verifies
                   that the register was cleared.
‘ADR:0x3’       -> While testing the ability for TClear to clear the registers, the register at
                   offset 0x3 is being verified.
'CNTREG 0'      -> Performing a walking 1's test on the address counter on board zero
'COUNT  0'      -> Performing the count up test on the address counter on board zero
```

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Mem_f

This selftest routine checks the ability to read and write to all of the memory locations on the EPIO board. There are two parts to this test. The first part checks the integrity of the address lines. This is done by writing incrementing data to all of the memory locations. This data is then read back, starting at address zero, and verified that it is still correct. The second part of the test checks the integrity of the data lines. This is done by performing a walking ones test on each of the memory chips that are installed.

Many of the other selftest routines require the use of the on board memory. Therefore, this test should pass before running any of the other selftest routines (except EPIO_dig_f, EPIO_status_f, EPIO_vclk_f, and EPIO_serial_f). The

Selftest fixture is not needed for this test.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------------
DMEM      0       TST: 3          ADR:Ox2       EXP:Ox02       ACT:Ox02
-------------------------------------------------------------------------------
```

Where 'DMEM 0' could be:

'DMEM 0'	-> Incrementing data test on the the Driver memory of board zero
'DWALK 0'	-> Walking 1's test on the Driver memory
'MMEM 0'	-> Incrementing data test on the Mask memory
'MWALK 0'	-> Walking 1's test on the Mask memory on board number zero
'RMEM 0'	-> Incrementing data test on the Result memory
'RWALK 0'	-> Walking 1's test on the Result memory on board number zero
'NRMEM 0'	-> Incrementing data test on the NML Result memory
'NRWALK0'	-> Walking 1's test on the NML Result memory on board number zero
'XNMEM 0'	-> Incrementing data test on the Expected NML memory
'XNWALK0'	-> Walking 1's test on the Expected NML memory on board number zero
'TSMEM 0'	-> Incrementing data test on the Timing Set Select memory
'TSWALK0'	-> Walking 1's test on the Timing Set Select memory on board number zero
'IMEM 0'	-> Incrementing data test on the Instruction memory
'IWALK 0'	-> Walking 1's test on the Instruction memory on board number zero
'JMEM 0'	-> Incrementing data test on the Jump memory
'JWALK 0'	-> Walking 1's test on the Jump memory on board number zero
‘ISMMEM’	-> Incrementing data test on the MSByte of the IO Formatter Segment Count memory
‘ISMWLK’	-> Walking 1’s test on the MSByte of the IO Formatter Segment Count memory
‘ISLMEM’	-> Incrementing data test on the LSByte of the IO Formatter Segment Count memory
‘ISLWLK’	-> Walking 1’s test on the LSByte of the IO Formatter Segment Count memory
‘DTSMEM’	-> Incrementing data test on the Driver Timing Set memory
‘DTSWLK’	-> Walking 1’s test on the Driver Timing Set memory
‘RTSMEM’	-> Incrementing data test on the Receiver Timing Set memory
‘RTSWLK’	-> Walking 1’s test on the Receiver Timing Set memory
‘ECSMEM’	-> Incrementing data test on the Execution Controller Segment Count memory
‘ECSWLK’	-> Walking 1’s test on the Execution Controller Segment Count memory
'TMMEM 0'	-> Incrementing data test on the Trigger Matrix Timing Set memory

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Status_f
This selftest routine is used to test the basic functionality of the START/STOP/CLOCK circuitry. It tests the ability of the functional calls to clear the STATUS bits coming from the EPIO Status register. From that point the routine determines whether or not the different STATUS bits (PRESTART, START, TGSTART, PRESTOP, STOP, TGSTOP, TEN, TESTEN, TGENABLE, and TIMEN) can be set correctly based on the circuit setup. Most of the bits are checked for both the conditions where they should be set and where they should not be set. A case where they should not be asserted would be if the software gives the strobe to assert the signal but a conditional signal is not yet asserted.

Since many of the other selftest routines (EPIO_f, EPIO_failadd_f, EPIO_format_f, EPIO_hsped_f, EPIO_instruc_f, EPIO_rmask_f, EPIO_timeset_f, EPIO_nml_f, EPIO_Inhibit_f, EPIO_tm_f, and EPIO_extsig_f) require proper operation of the START/STOP/CLOCK circuitry, this test must pass before executing those tests. The Selftest fixture is not needed for this test.

Sample Output

```
-------------------------------------------------------------------------------------
SETADD    0    TST: 1     ADR:Ox0     EXP:Ox00     ACT:Ox00
INCADD    0    TST: 2     ADR:Ox14    EXP:Ox14     ACT:Ox14
PSTRLO    0    TST: 3     ADR:Ox6A    EXP:Ox00     ACT:Ox00
PSTRH     0    TST: 4     ADR:Ox6A    EXP:Ox1000   ACT:Ox1000
STRLO     0    TST: 5     ADR:Ox6A    EXP:Ox1000   ACT:Ox1000
STRHI     0    TST: 6     ADR:Ox6A    EXP:Ox10DD   ACT:Ox10DD
TGSTLO    0    TST: 7     ADR:Ox6A    EXP:Ox00     ACT:Ox00
TGSTHI    0    TST: 8     ADR:Ox6A    EXP:OxD0     ACT:OxD0
PSTPLO    0    TST: 9     ADR:Ox6A    EXP:Ox00     ACT:Ox00
PSTPHI    0    TST: 10    ADR:Ox6A    EXP:Ox2000   ACT:Ox2000
STOPLO    0    TST: 11    ADR:Ox6A    EXP:Ox2000   ACT:Ox2000
STOPHI    0    TST: 12    ADR:Ox6A    EXP:Ox2022   ACT:Ox2022
TENLO1    0    TST: 13    ADR:Ox6A    EXP:Ox1000   ACT:Ox1000
TENHI     0    TST: 14    ADR:Ox6A    EXP:Ox10DD   ACT:Ox10DD
TENLO2    0    TST: 15    ADR:Ox6A    EXP:Ox3033   ACT:Ox3033
TGENLO    0    TST: 16    ADR:Ox6A    EXP:Ox1000   ACT:Ox1000
TGENHI    0    TST: 17    ADR:Ox6A    EXP:Ox10DD   ACT:Ox10DD
-------------------------------------------------------------------------------------
```

'SETADD' -> Tests ability to set the address counter
'INCADD' -> Tests ability to increment the address counter
'PSTRLO' -> Tests if the PRESTART status bit can be cleared
'PSTRHI' -> Tests if the PRESTART status bit can be set
'STRLO' -> Tests if the START status bit can be cleared
'STRHI' -> Tests if the START status bit can be set
'TGSTLO' -> Tests if the TGSTART status bit can be cleared
'TGSTHI' -> Tests if the TGSTART status bit can be set
'PSTPLO' -> Tests if the PRESTOP status bit can be cleared
‘PSTPHI' -> Tests if the PRESTOP status bit can be set
'STOPLO' -> Tests if the STOP status bit can be cleared
'STOPHI' -> Tests if the STOP status bit can be set
'TENLO1' -> Tests if the TESTENABLE status bit can be cleared
'TENHI' -> Tests if the TESTENABLE status bit can be set
'TENLO2' -> Tests if the TESTENABLE status bit can be reset after a halt
'TGENLO' -> Tests if the TGENABLE status bit can be cleared
'TGENHI' -> Tests if the TGENABLE status bit can be set

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Serial_f

This selftest routine is used to check the functionality of the EPIO’s serial interface. The EPIO selftest card includes a serial interface identical to the circuitry on an LFA card. The selftest card’s serial interface acts as a client to the EPIO’s serial interface. This test will also use the selftest card’s serial interface to the Selftest assembly’s motherboard.

In the first test, data is written to the selftest card through the Selftest assembly’s serial interface on its motherboard. This data is latched on the selftest card and then written to the EPIO using the serial interface through the Patchboard. The data read by the EPIO is then verified that it is correct. This test is done with the data equal to 0x00, 0xFF, as well as 0x01 through 0x80 with the lone high bit being shifted up for each test (for a total of ten tests).

In the second test, data is written to the selftest card through the Patchboard using the EPIO’s serial interface. This data is latched on the selftest card and then passed to the PC through the Selftest assembly’s serial interface on its motherboard. The data transferred to the PC is checked against the data that was written to the selftest card by the EPIO. This test is done with the data equal to 0x00, 0xFF, as well as 0x01 through 0x80 with the lone high bit being

shifted up for each test (for a total of ten tests).

This selftest should pass before running any of the other selftest routines that require the Selftest fixture (EPIO_f, EPIO_failadd_f, EPIO_format_f, EPIO_nml_f, EPIO_Inhibit_f, EPIO_hsped_f, EPIO_instruc_f, EPIO_rmask_f, EPIO_timeset_f, EPIO_tm_f, and EPIO_extsig_f). This is because any selftest routine that requires the Selftest fixture will need to serially write data to the selftest card in order to set it up for the that selftest.

Sample Output

```
-------------------------------------------------------------------------------------
EP->ST      0       TST: 7      ADR:Ox21F9800    EXP:Ox10    ACT:Ox10
ST->EP      0       TST: 13     ADR:Ox21F9800    EXP:Ox01    ACT:Ox01
-------------------------------------------------------------------------------------
'EP->ST 0'      -> Testing the EPIO to selftest serial communication on board 0
'ST->EP 0'      -> Testing the selftest to EPIO serial communication on board 0
```

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_f

This selftest routine is used to check the ability of the EPIO to drive and receive data with the differential drivers/receivers through the Patchboard. The routine verifies that all 32 channels can drive and receive data. The EPIO's serial interface through the Patchboard must be working properly in order for this test to pass. In addition, this selftest program needs a Selftest fixture with an EPIO selftest card in the same slot as the EPIO board.

The first test checks if the EPIO's differential drivers can drive data through the Patchboard. In this test, the selftest card is setup so it will latch the data driven to it. This data is then passed back to the EPIO using the EPIO's serial interface. The data received by the EPIO's serial interface is then checked against the data driven by the EPIO's differential drivers. This test is done with the data equal to 0x0000, 0xFFFF, as well as 0x01 through 0x8000 with the lone high bit being shifted up for each test (for a total of 34 tests).

The second test checks if the EPIO's differential receivers can accept data through the Patchboard. In this test, the data is written to the selftest card through the EPIO's serial interface. This data is latched on the selftest card and then driven back to the EPIO using the differential drivers on the selftest card. The EPIO is setup to read this data into its 'result' memory. The data stored in the 'result' memory is then checked against the data originally written to the selftest card through the EPIO's serial interface. This test is done with the data equal to 0x0000, 0xFFFF, as well as 0x01 through 0x8000 with the lone high bit being shifted up for each test (for a total of 34 tests).

Sample Output

```
-------------------------------------------------------------------------------------------
EPIODR    0    TST: 2     ADR:Ox21F9800    EXP:Ox00          ACT:Ox00
EPIORV    0    TST: 68    ADR:Ox21F9800    EXP:Ox80000000    ACT:Ox80000000
-------------------------------------------------------------------------------------------
'EPIODR 0'      -> Testing the ability of the EPIO to drive data through the Patchboard on
                   board 0
'EPIORV 0'      -> Testing the ability of the EPIO to receive data through the Patchboard on
                   board 0
```

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_HSpeed_f

This selftest routine tests the ability of the EPIO's receiver circuitry to accurately fill the result memory with a stream of data supplied by the drivers and the drive memory. The data stream does not pass through the Patchboard. The on-board connection between the drivers and receivers is used. The test is performed at selected clock rates over the range of the vector clock's capabilities. The EPIO_dig_f, EPIO_mem_f, and EPIO_status_f selftest routines must pass before running this test. The EPIO selftest card is not needed for this test.

First the EPIO's drivers and receivers are both enabled. The driver memory is then be filled up with known data. The known data will continuously cycle from zero to 255 within each of the four bytes that make up the driver memory. The vector clock will be started and the drivers will begin to output

their data. Once the clock stops, the data stored in the result memory will be verified. This test is performed over a range of vector clock frequencies.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-----------------------------------------------------------------------------------------
ADDR        0       TST: 1          ADR:Ox6D        EXP:Ox1F7       ACT:Ox1F7
100         0       TST: 11         ADR:Ox9         EXP:Ox9090909   ACT:Ox9090909
-----------------------------------------------------------------------------------------
'ADDR 0'        -> Checking the final address after executing the test at one clock rate on
                   board 0
'100 0'         -> Testing the vector clock rate of 100 Hz on board 0
```

- **If any failures occur during this Routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_RecMask_f

This selftest routine tests the ability of the EPIO receiver's evaluation circuitry to accurately mask received data. The drivers and the drive memory are used to supply the data used to test the evaluation circuitry. For this test, 16 channels will drive data through the Selftest Assembly to 16 different receive channels. The test is done in two different steps. The first step uses the lower 16 channels to drive data to the upper 16 channels. In the second step, the upper 16 channels drive data to the lower 16 channels. The test is performed at selected clock rates over the range of the vector clock's capabilities. The EPIO_HSpeed_f selftest routine must pass before running this test. The EPIO selftest card is needed for this test.

First, 16 EPIO drivers and 16 receivers are enabled. The driver memory is then filled up with known data. The known data will continuously cycle from zero to 255 within each of the four bytes that make up the driver memory. The receiver's evaluation circuitry is then setup so alternate receiver mask memory locations are filled with all zeros or all ones. This effectively alternately disables/enables the mask memory. By doing this, the routine verifies that the evaluation circuitry can run at the specified speed. The vector clock will be

started and the drivers will begin to output their data. Once the clock stops, the data stored in the result memory will be verified. For this test, the data stored should only match the driven data in every other memory location. The other locations will have been masked off (store data will be 0x00000000). This test is be performed over a range of vector clock frequencies.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------------------------
ADDR      0      TST: 1        ADR:Ox6D      EXP:Ox7FFFF      ACT:Ox7FFFF
10e6      0      TST: 11       ADR:Ox9       EXP:Ox9090909    ACT:Ox9090909
-------------------------------------------------------------------------------------------
'ADDR 0'        -> Checking the final address after executing the test at one clock rate on
                   board zero
'10e6 0'        -> Testing the vector clock rate of 10 MHz on board zero
```

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_ECLK_f

This selftest program is used to verify the accuracy of the DDS by measuring the Edge Clock against the certified TMS. The Edge Clock is generated from the DDS, either straight through, multiplied up, or divided down. The Edge Clock frequencies tested are selected to vary the DDS over its full range of operation. This range of DDS frequencies is 25 MHz to 50 MHz. Since the measurable frequencies are limited by the selftest and tester hardware, the range of Edge Clock frequencies tested vary from 1.57 MHz to 3.12 MHz. To obtain this range of measurable Edge Clock frequencies the 25-50 MHz DDS is divided down by 16.

The Edge Clock's output frequency is routed to the EXTCLK Patchboard pin. From there it's routed through the Selftest Assembly to an RMux card and finally to the TMS were the frequency is measured using the **freq()** functional call.

Sample Output

```
-------------------------------------------------------------------------------------
ECLK  0  TST: 5  HI   1.571E + 06 RDG   1.570E + 06 LO   1.569E + 06 HZ
-------------------------------------------------------------------------------------
```

The example line is testing the Edge Clock at 1.570 MHz.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_TM_f

This selftest routine is used to test some of the inputs and outputs of the Trigger Matrix on one EPIO board. This test consists of two separate sections. Each of the sections tests a different input or output. An output signal is defined as a signal that is generated on an EPIO and leaves through the Trigger Matrix. An input is defined as a signal that comes on to the board from the Trigger Matrix. The EPIO_hspeed_f selftest must pass before executing this test. A Selftest fixture is not needed for this test.

TEST #1: TMSIGx Output & TMSTOPTRG Input

This test checks the ability of the TMSIGx lines coming from the Instruction memory to stop a running test. There are four TMSIGx lines in the Instruction memory and they are numbered zero to three. A different TMSIGx bit is set at four different Instruction memory locations. The Trigger Matrix is setup such that only one of the TMSIGx outputs is routed to the TMSTOPTRG input signal. The STOP circuitry is setup to stop on the TMSTOPTRG signal from the Trigger Matrix. The test is started and it should stop when the specified bit set on the TMSIGx signal being tested is reached. After the test has stopped, the current address of the address counter is read. It is verified that this address is at the location were the TMSIGx bit was set.

TEST #2: TMSTART Output & TMSTOPTRG Input

The setup for this test is not how the TMSTART output and the TMSTOPTRG input would normally be setup to execute a test. It is only done to verify the operation of the TMSTART signal. The operation of the TMSTOPTRG input is tested in the first part of this selftest program. The Trigger Matrix is setup to route the TMSTART output to the TMSTOPTRG input. The STOP circuitry is

setup to stop on a signal from the Trigger Matrix. The START circuitry is setup to start on a strobe from the CPU. The test is then started. The START signal should propagate through the Trigger Matrix to the TMSTOPTRG signal and stop the test. It is then verified that the address counter is at the beginning of the test instead of being clocked to the end.

Sample Output

```
-------------------------------------------------------------------------------------------
TMSIG0    0    TST: 1    ADR:Ox32    EXP:Ox33    ACT:Ox33
ADDR      0    TST: 5    ADR:Ox0     EXP:Ox00    ACT:Ox00
STRTLO    0    TST: 6    ADR:Ox6A    EXP:Ox00    ACT:Ox00
STRTHI    0    TST: 7    ADR:Ox6A    EXP:Ox01    ACT:Ox01
SPADDR    0    TST: 8    ADR:OxFD    EXP:OxFD    ACT:OxFD
-------------------------------------------------------------------------------------------
'TMSIG0'      ->Testing Trigger Matrix signal #0, where the '0' could be 0-3.
'ADDR'        ->Verify the address counter did not start while testing the TMSTART signal.
'STRTLO'      ->Verify the START status bit is low while testing the TMSTART signal.
'STRTHI'      ->Verify the START status bit is high while testing the TMSTART signal.
'SPADDR'      ->Verify the address counter started while testing the TMSTART signal.
```

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_ExtSig_f

This selftest program is used to verify the operation of the external START and STOP signals that are supplied by receiver channels 16-23. The EPIO_hspeed_f selftest must pass before executing this test. A Selftest fixture is required for this test. There are two different sections to this selftest routine. Both are explained below. The external WAIT and JUMP signals are tested during EPIO_instruc_f.

TEST #1: External START (Receiver Channels #16 - 23)

This part of the test checks the Extrenal Start signal using receiver channels 16 - 23 as the external inputs. For this test, the selftest unit drives data to the EPIO board. Initially, the data driven and the polarity of the Start signal are configured so the test will not start. After the EPIO is ARMed and started, it is then verified that the MAR did not increment and that the Start Status bit is not asserted. Then, data is written to the selftest (and thus driven back to the EPIO) that will cause the test to start for the current polarity setting. The address counter is then verified that it incremented to where the UnConditional END instruction is and that the Start Status bit is asserted. This test is performed on

all eight External Input signals for both polarities and for both conditions were the test should and should not start.

TEST #2: External STOP (Receiver Channels #16 - 23)

This part of the test checks both of the External Stop signals using receiver channels 16 - 23 as the external inputs. For this test, the lower 16 channels drive to the upper 16 channels. Therefore, the EPIO selftest card is needed. The driver channel connected to receiver channel 16 - 23 is filled with the test data. The test is set up and then started. The driver data is clocked out and applied to the receiver channel being used as the external input. When the driven data matches the polarity of the Stop signal, the test will stop running. This test is performed on all eight external inputs, on both STOP signals, for both polarities, and for both conditions were it should stop and should not stop.

Sample Output

```
-------------------------------------------------------------------------------------------
ADS0-0    0    TST: 1     ADR:Ox0    EXP:Ox00        ACT:Ox00
SR0CL0    0    TST: 2     ADR:Ox6A   EXP:Ox00        ACT:Ox00
ADE0-0    0    TST: 3     ADR:Ox32   EXP:Ox00        ACT:Ox00
SRE0-0    0    TST: 4     ADR:Ox6A   EXP:Ox00        ACT:Ox00
RES0-2    0    TST: 13    ADR:Ox6A   EXP:OxFFFE0000  ACT:OxFFFE0000
-------------------------------------------------------------------------------------------
Testing the External START signal.
Where:
   'ADSe-s'     -> Means it is verifying the address before starting.
   'e'          -> External Input channel number being used to test the START signal
                   0 - External Input channel #16
                   1 - External Input channel #17
                   2 - External Input channel #18
                   3 - External Input channel #19
                   4 - External Input channel #20
                   5 - External Input channel #21
                   6 - External Input channel #22
                   7 - External Input channel #23
   's'          -> Test being performed:
                   0 - Don't allow the START signal to assert with the Polarity Low.
                   1 - Don't allow the START signal to assert with the Polarity High.
                   2 - Allow the START signal to assert with the Polarity Low.
                   3 - Allow the START signal to assert with the Polarity High.
   'SReCLs'     -> Means it is verifying the START status bit is cleared
   'ADEe-s'     -> Means it is verifying the test started by reading the address counter
   'SREe-s'     -> Means it is verifying the START status bit is set or cleared, depending on
                   the test being performed
```

'SREe-s' -> Means it is verifying that the data stored in Result memory is correct for the current External Input being tested

```
-------------------------------------------------------------------------------------
SP0CL0    0      TST: 33      ADR:Ox6A      EXP:Ox00      ACT:Ox00
ADD0-0    0      TST: 34      ADR:Ox32      EXP:OxFD      ACT:OxFD
-------------------------------------------------------------------------------------
```

Testing the External STOP signal.
Where:

'SPeCLs' -> Means it is verifying the STOP status bit is cleared before starting

'e' -> External STOP channel number being tested
- 0 - External Input channel #16
- 1 - External Input channel #17
- 2 - External Input channel #18
- 3 - External Input channel #19
- 4 - External Input channel #20
- 5 - External Input channel #21
- 6 - External Input channel #22
- 7 - External Input channel #23

's' -> Test being performed:
- 0 - Don't allow the STOP signal to assert with the Polarity Low.
- 1 - Don't allow the STOP sig to assert with the Polarity High.
- 2 - Allow the STOP signal to assert with the Polarity Low.
- 3 - Allow the STOP signal to assert with the Polarity Low.

'ADDe-s' -> Means it is verifying the test stoped at the correct address

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Instruc_f

This selftest routine is used to test the capabilities of the primary instructions. The instructions tested with this program are END, WAIT, DELAY, and JUMP. The Trigger Matrix secondary instructions are tested in the EPIO_tm_f selftest routine. The Enable Fail Checking secondary instruction is tested in the EPIO_failadd_f selftest routine. The Accumulator secondary instructions are tested as part of the EPIO_Carry_f selftest routine and the Utility Counter secondary instructions are tested as part of the DELAY primary instruction. The Selftest fixture is needed for this selftest.

The first group of tests verifies the functionality of the END instruction. The specific instructions included in this group are the Unconditional END, END on Equal, END on Unequal, and END Immediate instructions. To test the Unconditional END and END Immediate instructions, the END instruction is placed at a memory location. The test is started and when it stops, the memory location where the test stopped is verified that it is at the same location as the END instruction. The Unconditional END instruction is tested across multiple boards, if more than one board is present in the system.

The next tests are the END on Unequal and END on Equal tests. For each of these tests the driver and expected memory are filled with data that will cause the Equal flag to be set or reset based on if the Equal or Unequal END instruction is being tested. After each of these tests have stopped, it is verified that it stopped at the right address and that the Equal flag is set or cleared correctly for the test being done. In addition to testing, if the END instruction can stop a test, the edge clock's other two modes, lock and cycle, will also be tested with the END instruction. The lock and cycle modes are only valid for the END on Equal or END on Unequal instructions. The END on Equal and END on Unequal tests are performed across multiple boards if more than one board is present in the system.

The second group of tests verifies the functionality of the WAIT instruction. The first two tests in this section are WAIT on Equal and WAIT on Unequal. For each of these tests the driver and expected memory are filled with data that will cause the Equal flag to be set or reset based on if the Equal or Unequal WAIT instruction is being tested. The test is started and the address counter is read. The counter will be read repeatedly to verify that the test is waiting at the location of the WAIT on Equal or WAIT on Unequal instruction. Both of these tests are performed across multiple boards.

The next two tests verify the External WAIT instructions. The external WAIT signals used for these instructions come from receiver channels 16 - 23. Since the test is in the middle of a WAIT instruction, data can not be clocked out of the driver memory to the receiver channel to change the state of the External WAIT instruction. Therefore, data has to be passed to the selftest card to change the logic level on the receiver channel being used for the external WAIT signal while the test is waiting for it to change logic level. After the test is started, the program repeatedly reads the address counter and verifies that it is

waiting at the location where the External WAIT instruction is located. Once this is verified, the logic level of the receiver channel is changed by a serial write to the Selftest fixture. The address counter is then read to verify it is beyond the location where the External WAIT instruction is located. This test is only performed on one board at a time.

The third group of tests verifies the functionality of the DELAY on Counter instructions. The DELAY instruction is similar to WAIT. The difference is that this instruction is used to pause a test for a set time frame. The time frame is determined from the vector clock's frequency and the value loaded into the counter. For example, a delay of 5ms and a vector rate of 10MHZ would need a count value of 50000. For this test, the counter and vector rate are setup to cause a wait for a relatively long time frame (100's ms). The test is started and the address counter is read. The MAR will be read repeatedly to verify that the address counter is at the location of the WAIT instruction for approximately the same time frame as the delay period. This test is executed for a range of time delays, but only on one board at a time.

The fourth group of tests verifies the functionality of the JUMP instructions. The first test verifies the Unconditional JUMP instruction. For this test, the JUMP instruction will be placed at a certain location. An END instruction will be placed a few locations after this instruction and an END instruction will be placed at the location to be jumped to. Then, after the test stops, it verifies that the address counter is at the location jumped to and not at the location of the END instruction placed just after the JUMP instruction. This test is performed across multiple boards.

The next two tests verify the JUMP on Equal and JUMP on Unequal instructions. Each of these instructions are tested for the conditions where they should jump and where they should not jump. In all cases, an END instruction will be placed just after the JUMP and at the location jumped to. Each time the test is run, the address counter will be read. It will then be verified that the address is at either the jumped to location (if the test setup was such that the jump should have occurred) or just after the JUMP instruction (if the jump should not have occurred). This test is only performed on one board at a time.

For the JUMP on External tests, the EPIO will be set up to receive data from the Selftest fixture on the external channels (16 - 23). Data is written to the Selftest fixture to set the external bit. The test will then be started. After the test has

completed, it will be verified that the address counter is at the correct location. Again, these instructions will be checked for the locations where they should jump and should not jump. This test is only performed on one board at a time.

Primary instructions that have signals that are carried over the EPIO Ribbon Interconnection cable will be tested over multiple EPIO boards (if possible). By doing this, the integrity of the cable connections is verified. The instructions that are used to test multiple boards are the JUMP instruction and instructions that use the Equal flag.

Sample Output

```
-------------------------------------------------------------------------------------------
MUEND     0        TST: 1            ADR:Ox21F9800   EXP:Ox194     ACT:Ox194
-------------------------------------------------------------------------------------------
 Testing the UnConditional END Instruction (Master and Slaves)
      "MUEND"   - Current address on the Master board
      "SUEND"   - Current address on a Slave board

-------------------------------------------------------------------------------------------------
MEIM      0        TST: 2            ADR:Ox22A9600   EXP:Ox81      ACT:Ox81
-------------------------------------------------------------------------------------------------
 Testing the Immediate END Instruction (Master and Slaves)
      "MEIM" - Ending vector address on the Master board
      "SEIM" - Ending vector address on a Slave board

-----------------------------------------------------------------------------------------------
MCEUMS    0        TST: 2            ADR:Ox21F9800   EXP:Ox194     ACT:Ox194
TGRUN     0        TST: 3            ADR:Ox21F9800   EXP:Ox00      ACT:Ox00
TGSTOP    0        TST: 4            ADR:Ox21F9800   EXP:Ox00      ACT:Ox00
-------------------------------------------------------------------------------------------
 Testing the Conditional END Instruction (Master and Slaves)
      "MCEiMm"  - Current address on the master board
      "SCEiMm"  - Current address on a slave board
              'i' -> Conditional END instruction
                    'U' - END-On-UnEqual
                    'E' - END-On-Equal
              'm' -> End mode
                    'S' - Stop-On-END
                    'L' - Lock-On-END
                    'C' - Cycle-On-END
      "TGRUN"   - Timing Generator is still running on the master
      "TGSTOP"  - Timing Gernerator is stopped on the master
```

```
-------------------------------------------------------------------------------------
MWUM1F   0        TST: 20    ADR:Ox21F9800  EXP:Ox154    ACT:Ox154
MWEM2F   0        TST: 25    ADR:Ox21F9800  EXP:Ox303    ACT:Ox303
-------------------------------------------------------------------------------------
 Testing the Conditional WAIT Instruction (Master and Slaves)
     "MWiMmf"  - Current address on the Master board
     "SWiMmf"  - Current address on the Slave board
     "MWiMmE"  - Ending address on a Master for a test that
             should have waited but then was continued to the end.
     "SWiMmE"  - Ending address on a slave for a test that should
             have waited but then was continued to the end.
             'i' -> Conditional WAIT instruction
                   'U' - WAIT-On-UnEqual
                   'E' - WAIT-On-Equal
             'm' -> Test Method
                   1 - The test should wait at the test address
                   2 - The test should NOT wait at the test address
             'f' -> If there is an 'F' in this location it means this board was used to force the
                      Conditional WAIT. If this location is blank, this board was NOT used to
                      force the condition. This is true for both master and slave boards.

-------------------------------------------------------------------------------------
EW1M1L   0        TST: 26    ADR:Ox10       EXP:Ox204    ACT:Ox204
EW1MLE   0        TST: 27    ADR:Ox10       EXP:Ox300    ACT:Ox300
-------------------------------------------------------------------------------------
 Testing the External WAIT Instruction
     "EWiMmp"  - Current address of the board being tested
     "EWiMpE"  - Ending address for tests that should have
             waited but were then continued
             'i' -> WAIT-On-External instruction number
                   1 - WAIT-On-External signal #1
                   2 - WAIT-On-External signal #2
             'm' -> Test Method
                   1 - The test should wait at the test address
                   2 - The test should NOT wait at the test address
             'p' -> External signal polarity used for this test
                   L - Test was done with low polarity
                   H - Test was done with high polarity
      "ADR"      -> The value after ADR is the External Input channel being tested. It can
                      have a value of Ox10 to Ox17 representing channels #16 - 23.
```

```
-------------------------------------------------------------------------------------------
FWADD1   0      TST: 1     ADR:Ox200     EXP:Ox204     ACT:Ox204
FWNUM1   0      TST: 2     ADR:Ox200     EXP:Ox02      ACT:Ox02
FWA0T1   0      TST: 3     ADR:Ox200     EXP:Ox00      ACT:Ox00
FWA1T1   0      TST: 4     ADR:Ox200     EXP:Ox1FF     ACT:Ox1FF
-------------------------------------------------------------------------------------------
Testing the location of failures during a WAIT Instruction
        "FWADDm" - Address where the WAIT instruction is located
        "FWNUMm" - Number of expected failures for the current test setup
        "FWAiTm" - Address where the failure was logged at
                'm' -> Specifies which test setup was used. Will be a number
            from 1 to 7. Each test setup has the failure at a
            different address.
                'i' -> Index used to count through the number of expected
            failures. Will be a number from 0 to the number of failures displayed.
             Each test has a failure forced at the beginning of the vectors and
              within the pipeline of the WAIT instruction.


-------------------------------------------------------------------------------------------
D1V1     0      TST: 38          ADR:Ox21F9800   EXP:Ox0A      ACT:Ox0A
D1V1E    0      TST: 39          ADR:Ox21F9800   EXP:OxFFFF    ACT:OxFFFF
-------------------------------------------------------------------------------------------
 Testing the DELAY On Counter Instruction
     "DcVv"   - Number of times polled where the DELAY instruction is located
     "DcVvE"  - Ending address of the DELAY instruction test
             'c' -> Delay Counter being tested
                 1 - Delay Counter #1
                 2 - Delay Counter #2
             'v' -> Number representing the counter value being used
                 1 - 250000 (250 mSec)
                 2 - 500000 (500 mSec)
                 3 - 750000 (750 mSec)


-------------------------------------------------------------------------------------------
FDADD1   0      TST: 29    ADR:OxFFFF     EXP:OxFFFF    ACT:OxFFFF
FDNUM1   0      TST: 30    ADR:Ox6000     EXP:Ox02      ACT:Ox02
FDA0T1   0      TST: 31    ADR:Ox6000     EXP:Ox00      ACT:Ox00
FDA1T1   0      TST: 32    ADR:Ox6000     EXP:Ox5FFF    ACT:Ox5FFF
-------------------------------------------------------------------------------------------
Testing the location of failures during a DELAY Instruction
        "FDADDm" - Ending address of this test
        "FDNUMm" - Number of expected failures for the current test setup
        "FDAiTm" - Address where the failure was logged at
                'm' -> Specifies which test setup was used. Will be a number
            from 1 to 7. Each test setup has the failure at a
```

different address.
'i' -> Index used to count through the number of expected failures. Will be a number from 0 to the number of failures displayed. Each test has a failure forced at the beginning of the vectors and within the pipeline of the DELAY instruction.

```
-------------------------------------------------------------------------------------------
MUJUMP    0         TST: 50     ADR:Ox21F9800   EXP:Ox2509     ACT:Ox2509
-------------------------------------------------------------------------------------------
```

Testing the UnConditional JUMP Instruction (Master and Slaves)
"MUJUMP" - Master's address where the UnConditional JUMP was to
"SUJUMP" - Slave's address where the UnConditional JUMP was to

```
-------------------------------------------------------------------------------------------
MCJUM1    0         TST: 51     ADR:Ox21F9800   EXP:Ox2509     ACT:Ox2509
MCJEM2    0         TST: 54     ADR:Ox21F9800    EXP:Ox63      ACT:Ox63
-------------------------------------------------------------------------------------------
```

Testing the Conditional JUMP Instruction (Master and Slaves)
"MCJsMm" - Current address of the master board being tested
"SCJsMm" - Current address of a slave board being tested
's' -> Letter representing the conditional JUMP instruction being tested
U - JUMP-On-UnEqual
E - JUMP-On-Equal
'm' -> Test Method
1 - The test should jump to the Jump-To-Location
2 - The test should NOT jump

```
-------------------------------------------------------------------------------------------
JC1       0         TST: 55          ADR:Ox6A          EXP:Ox186A1    ACT:Ox186A1
JC1M1C    0         TST: 59          ADR:Ox21F9800   EXP:OxFFFFFF   ACT:OxFFFFFF
-------------------------------------------------------------------------------------------
```

Testing the JUMP Carry Clear On Counter Instrcution
"JCi" - Utility counter's reading used to verify the JCx instruction counted down
"JCiMmC" - Utility counters final reading of the board being tested
'i' -> Jump-On-Carry-Clear instruction
1 - JUMP-On-Carry-Clear Counter #1
2 - JUMP-On-Carry-Clear Counter #2
'm' -> Test Method
1 - The counter should count down
2 - The counter should NOT count down

```
-------------------------------------------------------------------------------------------
EJ1M1L      0        TST: 75            ADR:Ox10   EXP:Ox100A     ACT:Ox100A
-------------------------------------------------------------------------------------------
Testing the External JUMP Instrcution
      "EJiMmp" - Current address of the board being tested
            'i' -> Jump-On-External instruction
                  1 - JUMP-On-External #1
                  2 - JUMP-On-External #2
            'm' -> Test Method
                  1 - The test should jump to the Jump-To-Location
                  2 - The test should NOT jump
            'p' -> External signal polarity used for this test
                  L - Test was done with low polarity
                  H - Test was done with high polarity
      "ADR"       -> The value after ADR is the External Input channel being tested. It can
                     have a value of Ox10 to Ox17 representing channels #16 - 23.

-------------------------------------------------------------------------------------------
FJADD1      0        TST: 61    ADR:Ox1500     EXP:Ox3000     ACT:Ox3000
FJNUM1      0        TST: 62    ADR:Ox1500     EXP:Ox02       ACT:Ox02
FJA0T1      0        TST: 63    ADR:Ox1500     EXP:Ox00       ACT:Ox00
FJA1T1      0        TST: 64    ADR:Ox1500     EXP:Ox14FF     ACT:Ox14FF
-------------------------------------------------------------------------------------------
Testing the location of failures during a JUMP forward operation
          "FJADDm" - Ending address of this test
          "FJNUMm" - Number of expected failures for the current test setup
          "FJAiTm" - Address where the failure was logged at
                'm' -> Specifies which test setup was used. Will be a number
              from 1 to 7. Each test setup has the failure at a
              different address.
                'i' -> Index used to count through the number of expected
              failures. Will be a number from 0 to the number of
              failures displayed. Each test has a failure forced
              at the beginning of the vectors and within the
              pipeline of the JUMP instruction.

-------------------------------------------------------------------------------------------
FLADD1      0        TST: 88    ADR:Ox1000     EXP:Ox3000     ACT:Ox3000
FLNUM1      0        TST: 89    ADR:Ox1000     EXP:Ox66       ACT:Ox66
FLA0T1      0        TST: 90    ADR:Ox1000     EXP:Ox00       ACT:Ox00
FLA1T1      0        TST: 91    ADR:Ox1000     EXP:OxFFF      ACT:OxFFF
FLA2T1      0        TST: 92    ADR:Ox1000     EXP:OxFFF      ACT:OxFFF
FLA3T1      0        TST: 93    ADR:Ox1000     EXP:OxFFF      ACT:OxFFF
FLA4T1      0        TST: 94    ADR:Ox1000     EXP:OxFFF      ACT:OxFFF
FLA5T1      0        TST: 95    ADR:Ox1000     EXP:OxFFF      ACT:OxFFF
-------------------------------------------------------------------------------------------
```

Testing the location of failures during a looping JUMP operation
"FLADDm" - Ending address of this test
"FLNUMm" - Number of expected failures for the current test setup
"FLAiTm" - Address where the failure was logged at
'm' -> Specifies which test setup was used. Will be a number from 1 to 7. Each test setup has the failure at a different address.
'i' -> Index used to count through the number of expected failures. Will be a number from 0 to the number of failures displayed. Each test has a failure forced at the beginning of the vectors and within the loop created by the JUMP back instruction.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Timeset_f

This selftest routine is used to verify the resolution of a timing set. The Selftest fixture is required for this test.

For this test, the driver and receiver timing sets are setup to correctly clock data out from the driver memory and into the result memory. The data that was written into the result memory is verified. The active region of the driver's timing set is then changed by one edgeclock period for each test. For some of the tests, the correct data written to the result memory will not be the data driven out by the driver. This is because the driver and receiver timing sets are not in sync. By doing this, the ability of timing sets to correctly output and receive patterns can be verified as well as the ability to move the placement of the clock edges.

Sample Output

ADDR	0	TST: 1	ADR:Ox6D	EXP:Ox7FFFF	ACT:Ox7FFFF
TSET1	0	TST: 4	ADR:Ox20D9800	EXP:Ox2020000	ACT:Ox2020000

'ADDR 0' -> Verify the address counter incremented to the end on board number zero
'TSET1 0' -> Testing Timing Set configuration #1 on board number zero, where the '1' could be 0-2. Timing Set configuration numbers 0 and 1 will have the

driver and receiver timing sets configured such that they will correctly clock data out from the driver memory and into the result memory. Configuration #2 will have the receiver strobe after the active region of the driver data. Therefore, the driver data will not be strobed into the result memory.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Carry_f

This selftest routine is used to verify the ability of the algorithmic units to carry logic across one or more boards. The Selftest fixture is not needed for this test.

During this test, the accumulators will be setup so they will linearly add data (instead of looping on a few addresses). By doing this, all of the output vectors will be stored into the result memory. Therefore, all of the vectors can be read back and verified. This test will be performed for a multiple of test setups. One group of tests will have each of the eight bit accumulators on one board carry over to each of the other three accumulators on that board. Then, the second group of tests will have each accumulator on a board carry over to all four accumulators on a second board. This is done to verify the four global carry lines leaving each accumulator and going through the EPIO Ribbon Interconnect Cable.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------------
ADDOUT   0    TST: 1    ADR:Ox110   EXP:Ox113   ACT:Ox113
ADDIN    0    TST: 1    ADR:Ox110   EXP:Ox113   ACT:Ox113
O2 I3    0    TST: 2    ADR:Ox0     EXP:OxF0    ACT:OxF0
-------------------------------------------------------------------------------
```

'ADDOUT' -> Verify the ending address on the board with the Carry Out Accumulator

'ADDIN' -> Verify the ending address on the board with the Carry In Accumulator
'O2 I3' -> The Carry Out Accumulator is #2 and the Carry In Accumulator is #3.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Power to the Testhead and connections should always be checked.**

EPIO_FailAdd_f

This selftest routine is used to verify the fail address latch and the fail counter. The Selftest fixture is required for this test.

For this test, the driver and expected memory will be loaded with data that will cause failures. During this routine, the last 16 channels are used to drive the first 16 channels, and vice versa. The test will be run and the fail counter and latched failed addresses are read. The number of failures that occurred will be verified as well as the failed addresses. This test will be run several times. One of the tests will be setup so there will be no failures and the other tests will be run with a range of failures.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------
NF 0-1      0       TST: 2      ADR:Ox21F9800  EXP:Ox02     ACT:Ox02
FA 0-1      0       TST: 3      ADR:Ox21F9800  EXP:Ox32     ACT:Ox32
FD 0-1      0       TST: 4      ADR:Ox21F9800  EXP:OxCD     ACT:OxCD
-------------------------------------------------------------------------
```

'NF' -> Expected number of failures
'FA' -> Address were a failure occurred
'FD' -> Data at the failed address
'0-1' -> Byte #1 on board #0 is were the failure was forced to occur

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Power to the Testhead and connections should always be checked.**

EPIO_Format_f

This selftest routine is used to verify the output formats of a driver channel. The Selftest fixture is not needed for this test.

For this test, the timing set of a driver channel is setup so the active region is half of the vector clock period. The timing set for the receiver is first setup to latch the data during the driver's active region. The data will then be read and verified that it matches the driver data. Then, the receiver's timing set is setup to latch the data after the driver's active region. The data will then be read and verified that it is correct for the driver output format. This sequence is performed for all eight driver output formats.

Sample Output

```
R0-0V     0     TST: 1     ADR:Ox0     EXP:Ox00     ACT:Ox00
R0-1V     0     TST: 2     ADR:Ox1     EXP:Ox00     ACT:Ox00
```

Where 'R0' could be:
- 'R0' - Return-To-Zero
- 'R1' - Return-To-One
- 'RC' - Return-To-Complement
- 'NR' - No-Return
- '/R0' - Inverted Return-To-Zero
- '/R1' - Inverted Return-To-One
- '/RC' - Inverted Return-To-Complement
- '/NR' - Inverted No-Return

Where '-0V' could be:
- '-0V' - Driving a valid logic zero
- '-1V' - Driving a valid logic one
- '-0I' - Driving an invalid logic zero
- '-1I' - Driving an invalid logic one

The valid state means the receiver's timing set should latch the data during the driver's active region.
The invalid state means the receiver's timing set should latch the data during the driver's inactive region.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**

- **Power to the Testhead and connections should always be checked.**

EPIO_Inhibit_f

This selftest routine is used to test the Inhibit Fail mode. This mode is used to Inhibit a channel from causing an unequal condition and thus flagging a failure. This test checks each channel seperately.

For this test, 16 channels are configured to drive to the other 16 channels through the selftest. Each byte of driver and expected data is filled with incrementing data. Then, data in the channel driving to the test channel (the receiver channel) is modified to be unequal to the expected data on each vector. The first part of the test does not inhibit failures from ocuring. Therefore, after the test is run, there should be as many failures as there are vectors in the test. The received data is also verified that it is correct and it contains the opposite logic level than the expected on the test channel. Then for the second part of the test, the test channel is inhibited from flagging failures. The test is run again but this time there should be no failures logged. Once again the received data is verified that it contains the data opposite to the expected data on the test channel even though no failures were logged. This test is done for each channel as a receiver (or test channel).

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------------------------------------
ADDR    0    TST: 1      ADR:Ox6D       EXP:OxFF    ACT:OxFF
FAILS   0    TST: 2      ADR:Ox22A9600  EXP:OxFC    ACT:OxFC
INH-0   0    TST: 3      ADR:Ox0        EXP:Ox01    ACT:Ox01
ADDR    0    TST: 255    ADR:Ox6D       EXP:OxFF    ACT:OxFF
FAILS   0    TST: 256    ADR:Ox22A9600  EXP:Ox00    ACT:Ox00
INH-1   0    TST: 257    ADR:Ox0        EXP:Ox01    ACT:Ox01
-------------------------------------------------------------------------------------------------------
```

'ADDR' -> Verify the correct ending address for this test.
'FAILS' -> Verfify the correct number of fails occured for this test run.
'INH-0' -> Verify the correct data was written to the Result memory for the test that should log failures. The 'EXP' and 'ACT' data will be the expected and actual data stored to the result memory on the receive channels.
'INH-1' -> Verify the correct data was written to the Result memory for the test that

should not log failures. The 'EXP' and 'ACT' data will be the expected and actual data stored to the result memory on the receive channels.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Since data is being transfered through the Patchboard, there could be a Patchboard pin not making contact with the Selftest Assembly pin.**
- **Power to the Testhead and connections should always be checked.**

EPIO_nml_f

This selftest program is used to test the NML Detection capability of the EPIO. The NML Detection capability on the EPIO works on the principle that the LVDS receivers on the EPIO will drift high if they are not driven from an LFA or selftest card. There are two LVDS receivers per channel on an EPIO. One is the normal receiver. The other is for NML Detection. The two LVDS receivers are configured such that they will read opposite logic levels when a logic level is driven from the LFA or selfetst card. However, if an LVDS driver on the LFA or selftest card is disabled, the output from both receivers on the EPIO will drift high. The logic highs on both of these receivers on the EPIO flags a NML condition. In normal operation, the LFA disables the drivers to the EPIO when it detects a NML condition.

Since the selftest card does not have the circuitry that can generate various voltage levels, nor does it have the 'voltage window' detection capability like an LFA would have, it can not cause a NML condition. To simulate a NML condition, the control channels (28-31) which are used to control the channel direction on channels 0-27, can be used to disable the selftest's LVDS drivers to the EPIO. Therefore, the receivers on the EPIO will not be driven and both their outputs will drift high. The EPIO will then log this vector as being in NML.

For this test, channels 0-27 are configured as receivers and channels 28-31 are configured as drivers. The driver data behind the control channels (28-31) is filled with data that will enable one byte at a time to drive to the EPIO. After each byte has been allowed to drive for one vector (for a total of four vectors), all of the bytes will be disabled from driving to the EPIO for four vectors. This

eight vector pattern is repeated for the full memory depth. This pattern is used to just vary the disabling of the drivers to the EPIO.

It is highly recommended that this test only be executed when the Selftest Executive is configured to only output fail data. This is because the test performs millions of operations, and if all test data is output to the screen (or file) it would take hours to execute instead of minutes.

Sample Output

```
-------------------------------------------------------------------------------------------------------------
ADDR      0       TST: 1          ADR:Ox6D      EXP:Ox7FFFF    ACT:Ox7FFFF
NML       0       TST: 2          ADR:Ox0       EXP:OxFFFFF00  ACT:OxFFFFF00
RES       0       TST: 3          ADR:Ox0       EXP:OxFFFFF67  ACT:OxFFFFF67
-------------------------------------------------------------------------------------------------------------
```

'ADDR' -> Verify the correct ending address for this test.

'NML' -> Verify correct data was written to the NML Result memory. Any channel on the EPIO that is not driven to will store a logic one to memory. Any channel that is driven to will store a logic zero to memory. In this example, EPIO channels 0-7 were being driven to and the others detected the NML condition.

'RES' -> Verify correct data was written to the Result memory. Any channel on the EPIO that is not driven to will store a logic one to memory. Any channel that is driven to will store the data being driven from the selftest. In this example, EPIO channels 0 - 7 were being driven to and the data was 0x67, the other channels detected the NML condition.

- **If any failures occur during this routine, the problem is most likely on the EPIO board.**
- **Since the Selftest Assembly is required for this test, it could also cause the failures.**
- **Since data is being transfered through the Patchboard, there could be a Patchboard pin not making contact with the Selftest Assembly pin.**
- **Power to the Testhead and connections should always be checked.**

Appendix A - Glossary of Terms

Edge Clock - Locally generated clock that determines the placement resolution of the Timing Set edges (same frequency as the Master Clock).

Equal Flag - The wired-OR product of all the Evaluation Units in the system. It is used as a condition for several primary instructions, such as “End on Equal” or “Jump on Unequal”. If paired with the secondary instruction “Enable Fail Checking”, vectors that produce an unequal result are flagged as failures.

Evaluation Unit - The circuitry that combines a channel’s received data with its Expected Data and Mask Data. The results of all the Evaluation Units in a system are combined into the Equal Flag.

External Trigger and Qualifiers - a group of eight receivers on the Master EPIO Board that may also be used to trigger the start of the test, or as qualifiers for End, Wait and Jump instructions.

Master (Edge) Clock - The synchronizing clock of the EPIO System, distributed among all EPIO Boards.

Output Format - The method by which a timing pattern is applied to vector data. Several possibilities include Return-to Zero, Return-to-One, No-Return-(to-Zero), and Return-to-Compliment. Indirectly, Return-to-Z (high impedance) is also supported through the four control channels per board.

Test Counter Enable - Gate that allows the vector address counters to run. Without this enable, the test cannot advance. It is generated by the Execution Controller on the Master EPIO Board and is distributed among all EPIO Boards.

Timing Generator Enable - Gate that allows the Timing Generators to run. It is separate from the Test Enable to allow keep-alive clocks to continue after the test is over. It is generated by the Execution Controller on the Master EPIO Board and is distributed among all EPIO Boards.

Timing Pattern - a series of low and high states that determine active and inactive regions of driven output, or set the strobe edge of received data. Driver Timing Patterns usually begin and end with inactive regions (low bits) and have at least one active region between. A single pattern is used on a one channel for any single vector, although it may be used for any number of subsequent vectors.

Timing Segment - a section of a Timing Pattern that defines the state of the pattern for a specific number of edge clocks. The timing data may only change states between two segments. Timing Segments are programmed eight channels at a time, and include both driver and receiver timing data for each channel.

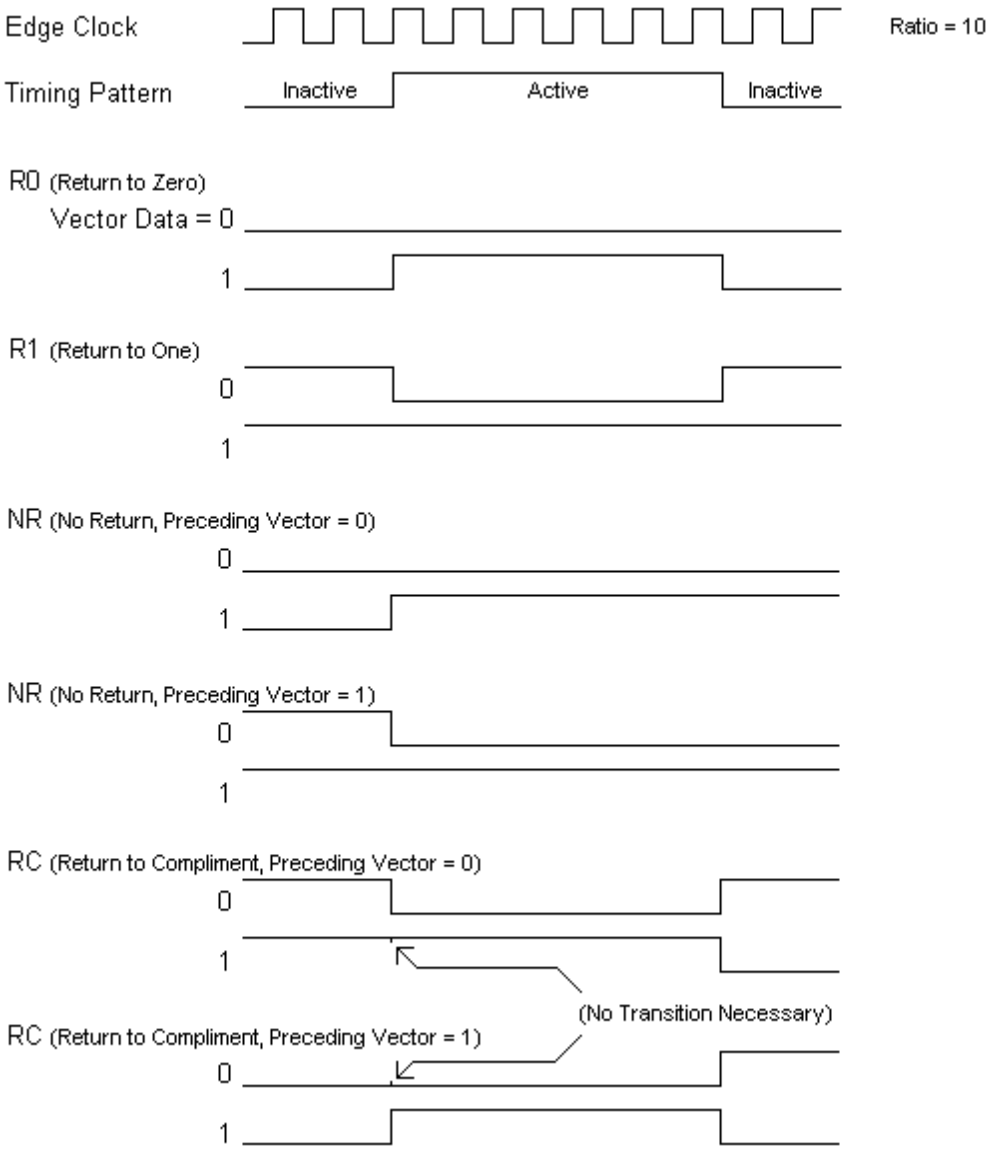
Timing Set - a group of Timing Patterns, one for every channel, related by the set number. For example when timing set five is selected, all the channels must use the pattern they designate as five. No mixing and matching of Patterns from different Sets is allowed.

Trigger Matrix - Global bus that allows dissimilar boards in the Testhead to synchronize events.

Vector Cycle - Locally generated signal that marks the end of a vector. The Timing Generators generate this signal at the end of each Timing Set. This signal allows the vector address counters to advance, and may have a variable frequency.

Appendix B - Driver Output Format

The following diagram illustrates the output expected given the example timing pattern and various output formats. A low and a high vector are provided for each case. Note that NR and RC formats are state machines; therefore, they depend on the previous vector for their beginning state.



Appendix C - Programming Rules

Timing Sets

Due to the manner in which Timing Sets are programmed into the EPIO system, the following is a short list of rules to be used when manually creating the data. The EPIO Pattern Editor handles many of these rules automatically.

1) Each Timing Set must have at least three segments
2) Each Timing Set must be a total of at least five segment counts long
3) Any segment must be at least one segment count long
4) Any segment may not be longer than 32768 segment counts long
5) The first segment of a Timing Set must start on a segment number divisible by four
6) All Timing Sets are programmed independently between bytes, subject to:
   a) The number of segments of a Timing Set may be variable among all bytes
   b) The total number of segment counts of a Timing Set among all bytes must be equal
   c) The first segment of a Timing Set must begin on the same set selection number among all bytes
7) The first segment of a Timing Set for NR or RC formatted driver channels may not be active (high)
8) The last segment of a Timing Set for receiver channels must be clear (triggered prior to that segment)

Appendix D - EPIO Vector Pipeline Stages

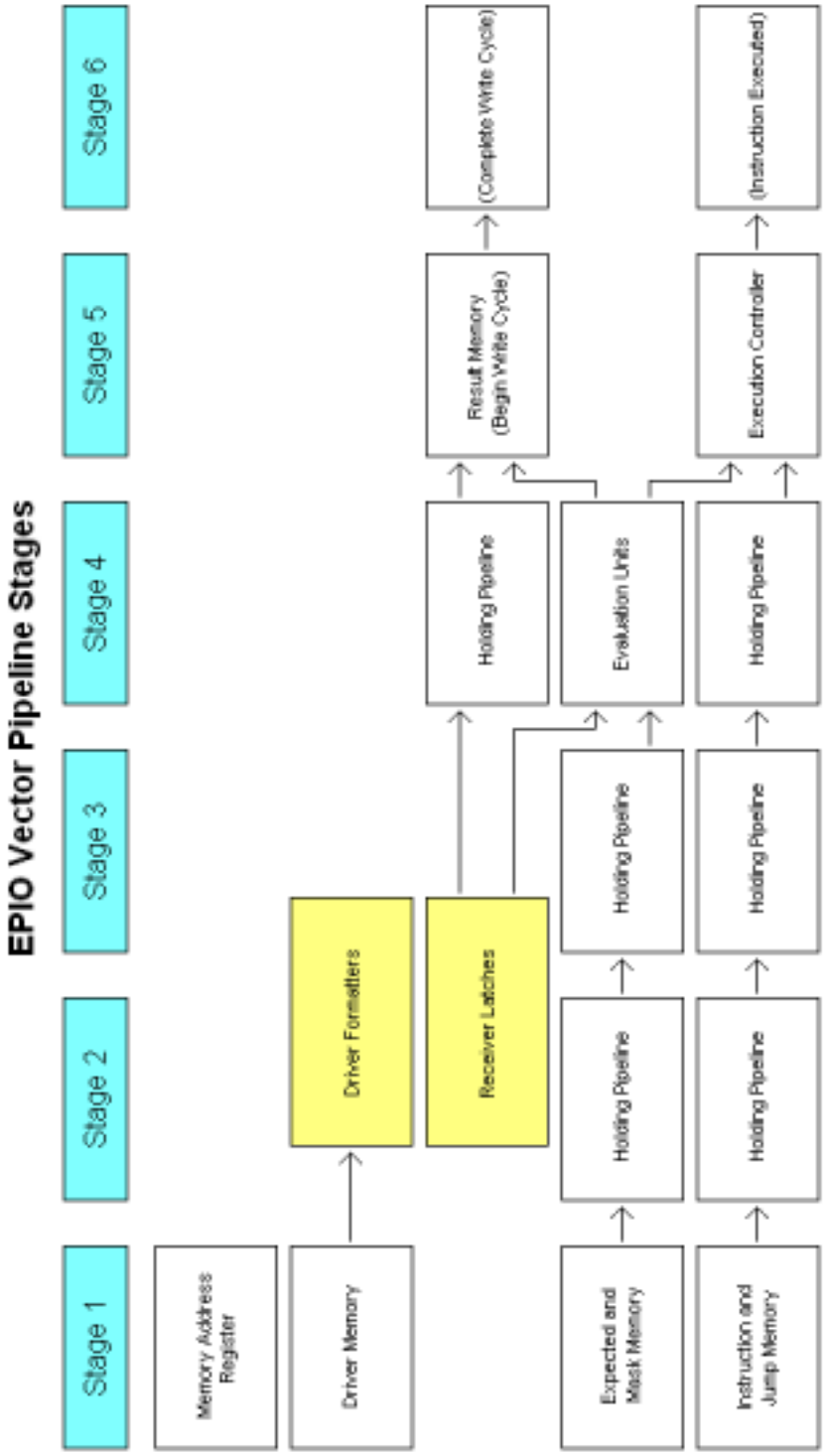
The overall length of the EPIO pipeline is six vector cycles, from beginning to end. Under many circumstances, the vectors run sequentially and the pipeline is not an issue. Functional calls automatically handle any data offsetting necessary when uploading and downloading test vectors to the hardware.

However, when the flow of a vector burst is altered with programmed instructions (such as: Wait, Delay, Jump, and End instructions), the behavior of any part of the system can be predicted based on the number of pipelined stages in front of the instruction unit (execution controller).

The six stages may be summarized as follows:

#	Name	Tasks Performed From Beginning of Cycle
1	Vector Memory Address Register	• Increment the vector memory address • (or) Load a new address (jump instruction) • Vector memory indexes data
2	Vector Data Latch	• Drive, Expected, Mask and Instruction data is latched into pipeline from vector memory • Drive data appears on outputs with Timing Sets applied (spans two stages)
3	Vector Data Continued	• Receive data is latched into holding area on programmed Timing Set strobes (in either Stage 2 or Stage 3.)
4	Evaluation Units	• Received Data is compared to Expected • Equal Flag is constructed
5	Execution Controller	• Results, Qualifiers, and Equal Flag are latched • Instructions are evaluated (Equal Flag or external Qualifiers may be used) • Valid Wait instructions will freeze all pipelines in their current positions (except for the received data pipeline)
6	Save Data	• Received data is written to vector memory • Instruction is executed • If a Jump is performed, new address is latched into the Memory Address Register (Stage 1)

As you may notice, the Execution Controller (Stage 5) is not on the same stage of the pipelines as the Vector Data Latch (Stage 2). While Stage 2 may have seemed like the natural place to put the instruction processing, we would not have been able to have conditional instructions. Because the Execution Controller is located behind the Evaluation Units (stage 4), we are able to have


EPIO Vector Pipeline Stages
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Stage 6
Memory Address Register
Driver Memory
Driver Formatters
Receiver Latches
Holding Pipeline
Holding Pipeline
Holding Pipeline
Holding Pipeline
Holding Pipeline
Evaluation Units
Holding Pipeline
Result Memory (Begin Write Cycle)
Execution Controller
(Complete Write Cycle)
(Instruction Executed)
Expected and Mask Memory
Instruction and Jump Memory

instructions that Wait, Jump, or End depending on the state of the received data.

Having the Execution Controller at Stage 5 creates some additional complexity, as you'll see below, but the benefits generally outweigh the tradeoffs.

End Instructions

End instructions are used to conditionally or unconditionally halt the running sequence of vectors. These instructions must pass through 5 pipeline stages before they are executed at the beginning of Stage 6. This is the point at which the Memory Address Register (MAR) will no longer increment, and the pipelines will no longer advance.

This means when the End instruction was in Stage 5 (before its execution), the data in that was in Stage 2 was the last vector driven. This creates a three-vector lead at the drivers. Therefore, it is highly recommended that three buffer vectors follow any End instruction to prevent undesired "garbage" vectors from appearing on the outputs of the system (with uncontrolled timing sets, as well).

Note that there is an unconditional "**End Immediate**" instruction that causes the vectors to stop without waiting for the instruction to pass through the long pipeline. This instruction executes at the beginning of Stage 3. Activity is stopped after the remainder of the current vector is sent to the drivers, and no buffer vectors are required.

Wait and Delay Instructions

Wait and Delay instructions are used to pause program flow (and the pipelines) on a specific vector until an event occurs. In the case of Wait, the event is external, expected from the product. In the case of Delay, an on-board counter is used to count a specific number of vectors. In either case, the drivers will pause on the data 3 vectors ahead of the Wait instruction vector. The receiver pipelines will not be paused, because incoming data is required for the conditional Wait instructions.

In the case of the "**Wait on Equal**" or "**Wait on Unequal**" instructions, the expected and mask data that appear on the instruction vector must be duplicated to the following vector. This is because the Evaluation Units (Stage

4) are not on the same stage as the Execution Controller (Stage 5). Once the Wait instruction has made it to Stage 5, the expected data now in the Evaluation Units (Stage 4) must not have changed; or else the Wait may be prematurely released.

If the failure detection is enabled on the wait vector, multiple failures may be recorded if the expected data does not match the received data (such as during a "**Wait on Unequal**" instruction.) Turn off fail detection to avoid this.

Jump Instructions

Jump instructions branch the flow of the program and may be used to construct loops and make decisions during the test. These instructions present special challenges in predicting system behavior, due to their branching nature. Because the MAR (Stage 1) is loaded with a new value, it takes several cycles for the data already in the pipeline to be completely flushed.

First, the instructions lag behind the address memory by 4 pipeline stages. When a jump is called, the address counter is already 4 addresses ahead of the execution controller. That means that the next four vector's worth of data has already been loaded into the pipeline. By the time the jump instruction reaches the execution controller, three of the four stages have already been driven to the product under test, and the fourth is inevitable.

The implications for loops are twofold. First, the Jump instruction is not the final vector of the loop. All vector data and instructions in the four vectors after the jump are going to be executed every iteration of the loop. If you follow a Jump instruction with an unconditional End instruction, the loop will be broken on its first pass. Because of this, the Jump instructions should not normally be followed by other instructions (within four vectors).

Second, no loop can be constructed smaller than five vectors. At minimum, the first vector would have the jump instruction and will be the target vector, and the next four vectors are for the pipeline depth. If a single vector loop is required, a Wait instruction might be adapted to work.

The EPStatus Functional Call

Calling **EPStatus** to find the current test position will return the current value in the Vector Memory Address Register (Stage 1). It will be one ahead of the vector currently at the driver outputs, and four ahead of the instruction

currently being executed.

At the end of a test, the Memory Address Register (MAR, Stage 1) will be five vectors ahead of the vector with the End instruction, and one vector ahead of the final vector driven. In the special case of a "Lock-on-End" mode, the MAR is prevented from clocking to its final value. The MAR value read will be four more than the End instruction vector, and will be the same as the final driven vector.

Received Data Memory

The Memory Address Register (MAR) is used to index both the driver memory and the receiver memory. Because of the pipeline, a vector's saved receiver data is offset four stages from the loaded driver data (Stage 6 and Stage 2, respectively). For the convenience of the user, we offset the downloaded results by four vectors. By using **EPGetResultData** and **EPGetNMLResultData** or the EPIO Pattern Editor, the user is presented with results that "line-up" with the drive data.

This is one of the only places where we make this concession, but it is not without side effects when combined with End and Jump instructions. First, the last three buffer vectors we recommend to be used after the End instruction will not appear to be filled with result data. This is due to the fact that the additional vectors will not flush the received data before the MAR is halted completely.

Illustration of Saved Vector Data After End Instruction
(Stop-on-End mode used)

MAR Value	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x40				
Instruction				End	– –	– –	– →	(Exec)				
Step Being Driven	0x32	0x33	0x34	0x35	0x36	0x37	0x38					} = MAR - 1
Save Location	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39				
Step Being Saved	0x28	0x29	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	
Download Location	0x28	0x29	0x30	0x31	0x32	0x33	0x34	0x35	?	?	?	= Save Location - 4

Second, Jump instructions cause the received data to appear in an unusual, but predictable, location. During a Jump instruction received data is written to memory, as must happen. However, the MAR target location is absolute and immediately follows the execution controller (Stage 5). This causes the data to not be written in the four vectors following the Jump instruction, but rather the four vectors starting with the Jump target vector. Considering that the

download software offsets the data by four vectors, it will **appear** that the four vectors prior to the Jump target location will be overwritten. This data is actually the data that matches the four vectors after the Jump instruction.

Illustration of Saved Vector Data Through Jump Instruction

MAR Value	0x33	0x34	0x35	0x36	0x37	0x70	0x71	0x72	0x73	0x74	0x75	0x76	
Instruction		J 0x70	– –	– –	– →	(Exec)							
Step Being Driven	0x32	0x33	0x34	0x35	0x36	0x37	0x70	0x71	0x72	0x73	0x74	0x75	= MAR - 1
Save Location	0x32	0x33	0x34	0x35	0x36	0x37	0x70	0x71	0x72	0x73	0x74	0x75	
Step Being Saved	0x28	0x29	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x70	0x71	
Download Location	0x28	0x29	0x30	0x31	0x32	0x33	0x66	0x67	0x68	0x69	0x70	0x71	= Save Location - 4

Appendix E - Implementation Examples

The following file is an example that implements a test generated with the EPIO Pattern Editor. The Pattern Editor creates and works with two file types: a configuration file and a vector file. Once both files have been debugged using the editor, a test executive is usually written that incorporates the necessary code to load, execute, and monitor the tests.

The following file assumes that the user interface is already included in the project, as well as the EPIO declaration module (epio32.bas). With a few changes to meet the specific needs of the test (saving result data or output to a log file, for example), this code could be quickly inserted into the executive environment.

The Code is written in Visual Basic; however, the algorithm used is generally applicable and recommended for other languages.

example.bas

```
Attribute VB_Name = “Module1”
Option Explicit

Public StopPressed As Boolean

Private Sub cmdStop_Click()
  StopPressed = True
End Sub

Public Sub ExecuteTest1(ByRef Failures As Long, ByRef ManualStop
As Boolean)
  ‘ Functional calls and the bits returned from EPStatus are
  ‘defined in the distributed file [Digalog]/include/epio32.bas
  ‘ Test Parameters
  Dim TotalBoardsUsed As Integer
  Dim MasterBoardNumber As Integer
  Dim TestLength As Long
  Dim PreLoad As Integer
  Dim EndMode As Integer

  ‘ Local Variables
  Dim BoardNumber As Long
  Dim TestDoneFlags As Long
```

```
  Dim TestDoneMask As Long
  Dim StatusWord As Long
  Dim ResultData() As Long
  Dim ResultNMLData() As Long
  Dim FailAddresses(0 To 255) As Long

  ‘ Setup Error handler and reset the stop switch
  On Error GoTo Test1ErrorHandler
  StopPressed = False

  ‘ If the function errors out, the number of failures returned
  ‘ will be -1
  Failures = -1

  ‘ Load the configuration file for this product and the vector
  ‘ file for Test1
  Call EPSetupConfig(“C:\Digalog\Projects\Project1\Product1.epc”)
  Call EPLoadVector(“C:\Digalog\Projects\Project1\Test1.epv”,
&H0&)

  ‘ The boards used, Master board, test length, Preload parameter,
  ‘ and EndMode parameter may also be parsed from the
  ‘ configuration file (above); however, this is beyond the scope
  ‘ of this example.
  TotalBoardsUsed = 4
  MasterBoardNumber = 0
  TestLength = &H3256&
  PreLoad = 0   ‘ Clear pipelines (do not advance data to product)
  EndMode = 0   ‘ Stop on End

  ‘ Arm three slave boards before the master board
  For BoardNumber = 0 To TotalBoardsUsed - 1
    If BoardNumber <> MasterBoardNumber Then
      Call EPArm(BoardNumber, &H0&, 0, PreLoad, EndMode)
    End If
  Next BoardNumber
  Call EPArm(MasterBoardNumber, &H0&, 1, PreLoad, EndMode)

  ‘ Manually start the test
  Call EPStart(MasterBoardNumber)

  ‘ Monitor the test’s progress
  TestDoneMask = esbTestStart Or esbTestStop Or esbTestGate Or
esbTestEnable
  TestDoneFlags = esbTestStart Or esbTestStop
  Do
    ‘ Check for a manual stop
```

```
    DoEvents
    If StopPressed Then
      Call EPHalt(MasterBoardNumber)
      Exit Do
    End If

    ‘ Get current test status
    ‘   These are the 7 valid cases for the flags as they progress
    ‘   through a test. Invalid combinations are not handled.
    Call EPStatus(MasterBoardNumber, 0, StatusWord)
    Select Case (StatusWord And TestDoneMask)
      Case 0
        Statusbar.Output “Waiting for Start”
      Case esbTestStart
        Statusbar.Output “Start Recevied”
      Case esbTestStart Or esbTestGate
        Statusbar.Output “Start Recevied”
      Case esbTestStart Or esbTestGate Or esbTestEnable
        Statusbar.Output “Running...”
      Case esbTestStart Or esbTestStop Or esbTestGate Or
esbTestEnable
        Statusbar.Output “Stop Received”
      Case esbTestStart Or esbTestStop Or esbTestEnable
        Statusbar.Output “Stop Received”
      Case esbTestStart Or esbTestStop
        Statusbar.Output “Stop Received”
        ‘ This is the exit condition
    End Select

  Loop Until (StatusWord And TestDoneMask) = TestDoneFlags

  ‘ Determine the fate of the Timing Generators
  Call EPStatus(MasterBoardNumber, 0, StatusWord)

  ‘ Check the Timing Generators for several possible states
  ‘ There may be errors, depending on the selected EndMode
  If StatusWord And esbTGEnable Then
    Select Case EndMode
      Case 0    ‘ Stop on End
        Statusbar.Output “ERROR: Timing Generators still running”
      Case 1    ‘ Lock on End
        Statusbar.Output “Timing Generators locked”
      Case 2    ‘ Cycle on End
        Statusbar.Output “Timing Generators cycling, waiting for
change ...”
        Do
          DoEvents
```

```
          If StopPressed Then
            Statusbar.Output “Timing Generators stopped manually”
            Call EPHalt(MasterBoardNumber)
            Exit Do
          End If
          Call EPStatus(MasterBoardNumber, 0, StatusWord)
        Loop Until (StatusWord And esbTGEnable) = 0
        Statusbar.Output “Timing Generators stopped”
    End Select
  Else
    Select Case EndMode
      Case 0    ‘ Stop on End
        Statusbar.Output “Timing Generators stopped”
      Case 1    ‘ Lock on End
        Statusbar.Output “ERROR: Timing Generators failed to lock”
      Case 2    ‘ Cycle on End
        Statusbar.Output “Timing Generators stopped”
    End Select
  End If

  ‘ Download results
  ReDim ResultData(0 To TestLength - 1) As Long
  ReDim ResultNMLData(0 To TestLength - 1) As Long
  For BoardNumber = 0 To TotalBoardsUsed - 1
    Call EPGetResultData(BoardNumber, &H0&, TestLength,
ResultData())
    Call EPGetResultNMLData(BoardNumber, &H0&, TestLength,
ResultNMLData())

    ‘ At this point, the data for this board may be saved or
processed,
    ‘ depending on need
  Next

  ‘ Check for failures and download
  If StatusWord And esbFail Then
    ‘The actual number of saved failures will be retrieved
    Failures = 255
    Call EPFailAddresses(MasterBoardNumber, Failures,
FailAddresses())

    ‘ At this point, the list of failed addressed may be saved or
processed,
    ‘ depending on need
  Else
    Failures = 0
```

```
  End If

Test1ErrorHandler:
  ‘ Did an error bring the program here?
  Select Case Err.Number
    Case 0
      ‘ No error
    Case Is > 98 * 256
      ‘ Error most likely returned from Digalog libraries
      ‘   (format the number into Digalog standard)
      Statusbar.Output “Error Encountered (“ & CStr(Err.Number \
256) & “:” & _
                      CStr(Err.Number Mod 256) & “)”
    Case Else
      ‘ Error most likely retuened from VB runtime engine
      ‘   (description available)
      Statusbar.Output “Error Encountered (“ & CStr(Err.Number) &
“) - “ & _
                      Err.Description
  End Select

  ‘ Return ManualStop if the test was aborted
  ManualStop = StopPressed
End Sub
```

Appendix F - Error Codes

108:107	(EPIO))	A call to EPArm() was made while the master board was in a Lock-On_End condition. Use either EPArmDuringLock() or call EPHalt().
108:121	(EPIO)	While reading the Configuration file, the number of voltage levels listed under 'LFAVoltageBytex' did not match the expected number.
108:122	(EPIO)	While reading the Configuration file, the 'LFAVoltageBytex_Key' could not be found.
108:123	(EPIO)	While reading the Configuration file, the '[Customer]' section could not be found.
108:124	(EPIO)	The Timing Set data for one line read from the 'epc' file did not contain the proper number of segment counts for the number of boards.
108:125	(EPIO)	While reading the Configuration file, the 'StartSegmentx" key was found but there was an invalid value assigned to it.
108:126	(EPIO)	Whiile reading the Configuration file, the 'NumberOfTimingSets' key was found but there was an invalid value assigned to it.
108:128	(EPIO)	Given EPIO board number does not exist.
108:129	(EPIO)	Illegal EPIO address for an 'Offset' or 'StartAddr'.
108:130	(EPIO)	The requested number of failed addresses is too many.
108:131	(EPIO)	Illegal 'MasterBd' parameter in EpArm().
108:132	(EPIO)	Illegal 'ByteNum' parameter in EpCarryMatrixSetup().
108:133	(EPIO)	Illegal 'CarryIn' parameter in EpCarryMatrixSetup().
108:134	(EPIO)	Illegal 'CarryOut' parameter in EpCarryMatrixSetup().
108:135	(EPIO)	In EpChannelSetup(), a nibble is defined as bi-dir but the last nibble is not a fixed driver.
108:136	(EPIO)	In EpChannelSetup(), a nibble is defined as bi-dir but the last nibble is a receiver.
108:137	(EPIO)	Illegal length for the available EPIO memory.
108:138	(EPIO)	Illegal 'Channel' parameter in EpDriverFormat().
108:139	(EPIO)	Illegal 'OutputFormat' parameter in EpDriverFormat().
108:140	(EPIO)	Illegal 'ClockSource' parameter in EpEdgeClock().
108:141	(EPIO)	Illegal 'Frequency' parameter in EpEdgeClock().
108:142	(EPIO)	Illegal 'EndMode' parameter in EpEdgeClock().
108:143	(EPIO)	Illegal 'Delay' parameter in EpEdgeDelay().
108:144	(EPIO)	Illegal 'Selection' parameter in EpStatus().
108:145	(EPIO)	Illegal 'BurstCount' parameter in EpStrobe().
108:146	(EPIO)	Illegal 'Ratio' parameter in EpVectorRatio().
108:147	(EPIO)	Illegal 'StartSignal' parameter in EpVectorGate().
108:148	(EPIO)	Illegal 'StartPolarity' parameter in EpVectorGate().
108:149	(EPIO)	Illegal 'StopSignal' parameter in EpVectorGate().
108:150	(EPIO)	Illegal 'StopPolarity' parameter in EpVectorGate().
108:151	(EPIO)	Illegal 'StartQualifierEnable' parameter in EpVectorGate().
108:152	(EPIO)	Illegal 'StartQualifierPolarity' parameter in EpVectorGate().
108:153	(EPIO)	Illegal 'Preload' parameter in EpArm().
108:154	(EPIO)	Illegal 'EndMode' parameter in EpArm().
108:155	(EPIO)	The timing generator is still running.
108:156	(EPIO)	The Edge Clock generator did not lock.

108:157	(EPIO)	Illegal Adjusted Edge Clock frequency.
108:158	(EPIO)	The given EPIO board is not the master.
108:159	(EPIO)	The given EPIO board is not armed.
108:160	(EPIO)	The TG Start status bit was never asserted.
108:161	(EPIO)	The Stop status bit was never asserted.
108:162	(EPIO)	There are not enough EPIO boards with termination.
108:163	(EPIO)	There are too many EPIO boards with termination.
108:164	(EPIO)	The EPIO boards with termination must be on the ends.
108:165	(EPIO)	The Vector clock is still running.
108:166	(EPIO)	The Stop bit was never set.
108:167	(EPIO)	The Start status bit was never asserted.
108:168	(EPIO))	Could not find an EPIO board in the system.
108:169	(EPIO)	The Edge Clock control register did not verify.
108:170	(EPIO)	The Status register did not verify in EpArm().
108:171	(EPIO)	The Carry Matrix control register did not verify.
108:172	(EPIO)	The Fixed Driver control register did not verify.
108:173	(EPIO)	The Bi-Directional Enable register did not verify.
108:174	(EPIO)	The Driver Format control register did not verify.
108:175	(EPIO)	The DDS Frequency control register did not verify.
108:176	(EPIO)	The Edge Clock Delay control register did not verify.
108:177	(EPIO)	The Vector Clock control register did not verify.
108:178	(EPIO)	The Selftest control register did not verify.
108:179	(EPIO)	The Gate control register did not verify.
108:180	(EPIO)	The External Polarity control register did not verify.
108:181	(EPIO)	The External Inverse control register did not verify.
108:182	(EPIO)	During a write/readback operation to the I/O Formatter Control register, the value read back did not match the value written.
108:183	(EPIO)	Illegal External Jump Qualifier Polarity parameter in EpExternalQualPol().
108:184	(EPIO)	Illegal External Wait Qualifier Polarity parameter in EpExternalQualPol().
108:185	(EPIO)	Illegal External End Qualifier Polarity parameter in EpExternalQualPol().
108:186	(EPIO)	The EPIO is not in the Lock-On-End mode when the EpArmDuringLock() call was made.
108:187	(EPIO)	The 16 Bit Register on the EPIO Selftest card did not verify.
108:188	(EPIO)	The version value read is illegal or not understood by this software.
108:189	(EPIO)	EpSetupConfig() MUST be called before calling EpLoadVector().
108:190	(EPIO)	While reading the Configuration file, no 'NumTimingSets' key was found.
108:191	(EPIO)	While reading the Configuration file, no 'SegmentCountsx' key was found.
108:192	(EPIO)	While reading the Configuration file, no 'StartSegmentx' key was found.
108:193	(EPIO)	While reading the Configuration file, no '[TimingSets]' section was found.
108:194	(EPIO)	An illegal byte number was given in EPIOTimingData().
108:195	(EPIO)	An illegal segment count was given in either EPIOTimingData() or EPECTimingData().
108:196	(EPIO)	The label parameter in EPGetLabelStep is not a valid string.
108:197	(EPIO)	The program could not open the Configuration file created by the EPIO Editor.
108:198	(EPIO)	The program could not open the Vector file created by the EPIO Editor.

108:199	(EPIO)	The section name is invalid.
108:200	(EPIO)	A line was read from the Configuration file that contained brackets '[ ]' but no section name was between them.
108:201	(EPIO)	While reading the Configuration file, two sections with the same name were found.
108:202	(EPIO)	While reading the Configuration file, two keys with the same name were found.
108:203	(EPIO)	While reading the Configuration file, two labels with the same name were found.
108:204	(EPIO)	While reading the Configuration file, no [Config] section was found.
108:205	(EPIO)	While reading the Configuration file, no EPIOConfiigFileVersion key was found.
108:206	(EPIO)	The version read from the Configuration file is not recognized by this software.
108:207	(EPIO)	While reading the Configuration file, no NumberOfBoards key was found.
108:208	(EPIO)	While reading the Configuration file, no BoardsUsed key was found.
108:209	(EPIO)	While reading the Configuration file, the BoardsUsed key was found but there were no boards assigned to it.
108:210	(EPIO)	While reading the Configuration file, the NumberOfBoards value did not match the number of boards listed under BoardsUsed.
108:211	(EPIO)	While reading the Configuration file, the NumberOfBoards key was found but no value was assigned to it.
108:212	(EPIO)	While reading the Configuration file, no Master key was found.
108:213	(EPIO)	While reading the Configuration file, the Master key was found but no value was assigned to it.
108:214	(EPIO)	While reading the Configuration file, no VectorRatio key was found.
108:215	(EPIO))	While reading the Configuration file, the VectorRatio key was found but no value was assigned to it.
108:216	(EPIO)	While reading the Configuration file, no ClockSource key was found.
108:217	(EPIO)	While reading the Configuration file, the ClockSource key was found but no value was assigned to it.
108:218	(EPIO)	While reading the Configuration file, no Frequency key was found.
108:219	(EPIO)	While reading the Configuration file, the Frequency key was found but no value was assigned to it.
108:220	(EPIO)	While reading the Configuration file, no StartTriggers key was found.
108:221	(EPIO)	While reading the Configuration file, the StartTriggers key was found but no value was assigned to it.
108:222	(EPIO)	While reading the Configuration file, no StopTriggers key was found.
108:223	(EPIO)	While reading the Configuration file, the StopTriggers key was found but no value was assigned to it.
108:224	(EPIO)	While reading the Configuration file, no StopQualifiers key was found.
108:225	(EPIO)	While reading the Configuration file, the StopQualifiers key was found but no value was assigned to it.
108:226	(EPIO)	While reading the Configuration file, no JumpQualifiers key was found.
108:227	(EPIO)	While reading the Configuration file, the JumpQualifiers key was found but no value was assigned to it.
108:228	(EPIO)	While reading the Configuration file, no WaitQualifiers key was found.

108:229	(EPIO)	While reading the Configuration file, the WaitQualifiers key was found but no value was assigned to it.
108:230	(EPIO)	While reading the Configuration file, no BoardsDriverEnable key was found.
108:231	(EPIO)	While reading the Configuration file, no BoardsReceiverEnable key was found.
108:232	(EPIO)	While reading the Configuration file, the BoardsDriverEnable key was found but no value was assigned to it.
108:233	(EPIO)	While reading the Configuration file, the BoardsReceiverEnable key was found but no value was assigned to it.
108:234	(EPIO)	While reading the Configuration file, no CarryIn key was found.
108:235	(EPIO)	While reading the Configuration file, the CarryIn key was found but no value was assigned to it.
108:236	(EPIO)	While reading the Configuration file, no CarryBus key was found.
108:237	(EPIO)	While reading the Configuration file, the CarryBus key was found but no value was assigned to it.
108:238	(EPIO)	While reading the Configuration file, no [Channels] section was found.
108:239	(EPIO)	While reading the Configuration file, no channel number was found.
108:240	(EPIO)	While reading the Configuration file, the channel number was found but the value read was not recognized by this software, or there was no value.
108:241	(EPIO)	While reading the Configuration file, no [Timing] section was found.
108:242	(EPIO)	While reading the Configuration file, the number of lines associated with the [Timing] section does not match the size of the Timing Set memory.
108:243	(EPIO)	While reading the Configuration file, the length of one Timing data line does not match the number of boards.
108:244	(EPIO)	While reading the Configuration file, the linked list of data is corrupted.
108:245	(EPIO)	While reading the Vector file, a section header could not be read.
108:246	(EPIO)	While reading the Vector file, a section could not be read.
108:247	(EPIO)	While reading the Vector file, a header was read but the header's title did not match what was expected.
108:248	(EPIO)	While reading the Vector file, the requested number of vectors for one of the sections could not be read.
108:249	(EPIO)	While reading the Vector file, the number of vectors read does not match the number expected.
108:250	(EPIO)	While reading the Vector file, the number of total vectors read for a dboard does not match the number specified.
108:251	(EPIO)	While reading the Vector file, the version read from the file is not recognized by this software.
108:252	(EPIO)	While reading the Vector file, the number of bytes read from the Configuration file did not match what was expected.
108:253	(EPIO)	While reading the Vector file, the number of bytes read from the label, operand, comment section did not match what was expected.
108:254	(EPIO)	While reading the Vector file, the number of bytes read from the EPIO Vector file did not match what was expected.
108:255	(EPIO)	The labels have not been configured into a table yet. The EPLoadVector call must be made, and then the EPGetLabelStep call.